Users' Manual

Version 1.1
March 10, 2015

BRAID

PARALLEL MULTIGRID IN TIME

V. A. Dobrev, R. D. Falgout, Tz. V. Kolev, N. A. Petersson, J. B. Schroder, U. M. Yang
Center for Applied Scientific Computing (CASC)
Lawrence Livermore National Laboratory

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-SM-660398

Contents

1 Abstract

This package implements an optimal-scaling multigrid solver for the (non)linear systems that arise from the discretization of problems with evolutionary behavior. Typically, solution algorithms for evolution equations are based on a time-marching approach, solving sequentially for one time step after the other. Parallelism in these traditional time-integration techniques is limited to spatial parallelism. However, current trends in computer architectures are leading towards systems with more, but not faster, processors, i.e., clock speeds are stagnate. Therefore, faster overall runtimes must come from greater parallelism. One approach to achieve parallelism in time is with multigrid, but extending classical multigrid methods for elliptic operators to this setting is a significant achievement. In this software, we implement a non-intrusive, optimal-scaling time-parallel method based on multigrid reduction techniques. The examples in the package demonstrate optimality of our multigrid-reduction-in-time algorithm (MGRIT) for solving a variety of equations in two and three spatial dimensions. These examples can also be used to show that MGRIT can achieve significant speedup in comparison to sequential time marching on modern architectures.

It is **strongly recommended** that you also read `Parallel Time Integration with Multigrid` after reading the Overview of the XBraid Algorithm. It is a more in depth discussion of the algorithm and associated experiments.

2 Introduction

2.1 Meaning of the name

We chose the package name XBraid to stand for *Time-Braid*, where X is the first letter in the Greek work for time, *Chronos*. The algorithm *braids* together time-grids of different granularity in order to create a multigrid method and achieve parallelism in the time dimension.

2.2 Advice to users

The field of parallel-in-time methods is in many ways under development, and success has been shown primarily for problems with some parabolic character. While there are ongoing projects (here and elsewhere) looking at varied applications such as hyperbolic problems, computational fluid dynamics, power grids, medical applications, and so on, expectations should take this fact into account. That being said, we strongly encourage new users to try our code for their application. Every new application has its own issues to address and this will help us to improve both the algorithm and the software.

2.3 Overview of the XBraid Algorithm

The goal of XBraid is to solve a problem faster than a traditional time marching algorithm. Instead of sequential time marching, XBraid solves the problem iteratively by simultaneously updating a space-time solution guess over all time values. The initial solution guess can be anything, even a random function over space-time. The iterative updates to the solution guess are done by constructing a hierarchy of temporal grids, where the finest grid contains all of the time values for the simulation. Each subsequent grid is a coarser grid with fewer time values. The coarsest grid has a trivial number of time steps and can be quickly solved exactly. The effect is that solutions to the time marching problem on the coarser (i.e., cheaper) grids can be used to correct the original finest grid solution. Analogous to spatial multigrid, the coarse grid correction only *corrects* and *accelerates* convergence to the finest grid solution. The coarse grid does not need to represent an accurate time discretization in its own right. Thus, a problem with many time steps (thousands, tens of thousands or more) can be solved with 10 or 15 XBraid iterations, and the overall time to solution can be greatly sped up. However, this is achieved at the cost of more computational resources.

To understand how XBraid differs from traditional time marching, consider the simple linear advection equation, $u_t = -cu_x$. The next figure depicts how one would typically evolve a solution here with sequential time stepping. The initial condition is a wave, and this wave propagates sequentially across space as time increases.

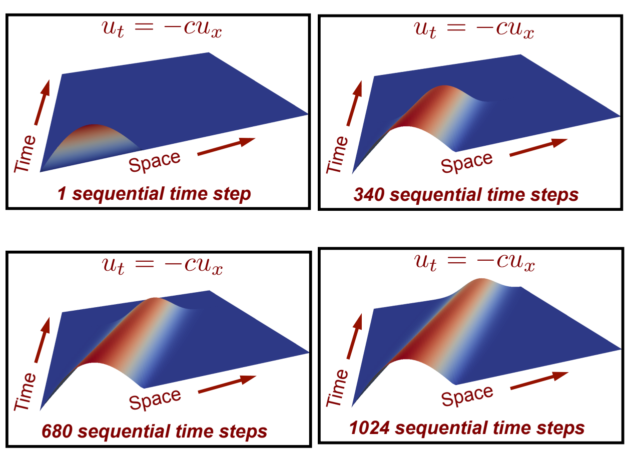


Figure 1: Sequential time stepping.

XBraid instead begins with a solution guess over all of space-time, which for demonstration, we let be random. An XBraid iteration does

1. Relaxation on the fine grid, i.e., the grid that contains all of the desired time values Relaxation is just a local application of the time stepping scheme, e.g., backward Euler.
2. Restriction to the first coarse grid, i.e., interpolate the problem to a grid that contains fewer time values, say every second or every third time value.
3. Relaxation on the first coarse grid
4. Restriction to the second coarse grid and so on...
5. When a coarse grid of trivial size (say 2 time steps) is reached, it is solved exactly.
6. The solution is then interpolated from the coarsest grid to the finest grid

One XBraid iteration is called a *cycle* and these cycles continue until the solution is accurate enough. This is depicted in the next figure, where only a few iterations are required for this simple problem.

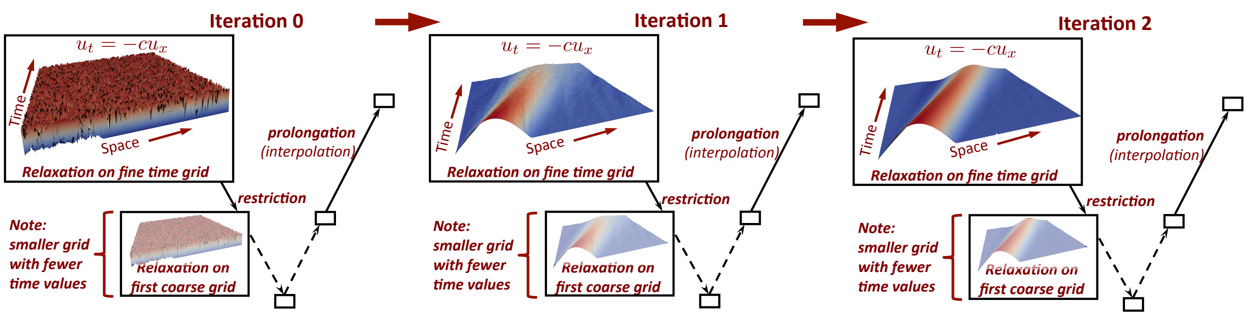


Figure 2: XBraid iterations.

There are a few important points to make.

- The coarse time grids allow for global propagation of information across space-time with only one XBraid iteration. This is visible in the above figure by observing how the solution is updated from iteration 0 to iteration 1.
- Using coarser (cheaper) grids to correct the fine grid is analogous to spatial multigrid.
- Only a few XBraid iterations are required to find the solution over 1024 time steps. Therefore if enough processors are available to parallelize XBraid, we can see a speedup over traditional time stepping (more on this later).
- This is a simple example, with evenly space time steps. XBraid is structured to handle variable time step sizes and adaptive time step sizes, and these features will be coming.

To firm up our understanding, let's do a little math. Assume that you have a general system of ordinary differential equations (ODEs),

$$u'(t) = f(t, u(t)), \quad u(0) = u_0, \quad t \in [0, T].$$

Next, let $t_i = i\delta t, i = 0, 1, ..., N$ be a temporal mesh with spacing $\delta t = T/N$, and u_i be an approximation to $u(t_i)$. A general one-step time discretization is now given by

$$\begin{aligned} u_0 &= g_0 \\ u_i &= \Phi_i(u_{i-1}) + g_i, \quad i = 1, 2, ..., N. \end{aligned}$$

Traditional time marching would first solve for $i = 1$, then solve for $i = 2$, and so on. For linear time propagators $\{\Phi_i\}$, this can also be expressed as applying a direct solver (a forward solve) to the following system:

$$A\mathbf{u} \equiv \begin{pmatrix} I & & & \\ -\Phi_1 & I & & \\ & \ddots & \ddots & \\ & & -\Phi_N & I \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_N \end{pmatrix} = \begin{pmatrix} g_0 \\ g_1 \\ \vdots \\ g_N \end{pmatrix} \equiv \mathbf{g}$$

or

$$A\mathbf{u} = \mathbf{g}.$$

This process is optimal and O(N), but it is sequential. XBraid achieves parallelism in time by replacing this sequential solve with an optimal multigrid reduction iterative method [1] applied to only the time dimension. This approach is

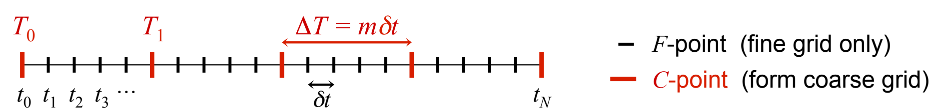
- nonintrusive, in that it coarsens only in time and the user defines Φ. Thus, users can continue using existing time stepping codes by wrapping them into our framework.
- optimal and O(N), but O(N) with a higher constant than time stepping. Thus with enough computational resources, XBraid will outperform sequential time stepping.
- highly parallel

We now describe the two-grid process in more detail, with the multilevel analogue being a recursive application of the process. We also assume that Φ is constant for notational simplicity. XBraid coarsens in the time dimension with factor $m > 1$ to yield a coarse time grid with $N_\Delta = N/m$ points and time step $\Delta T = m\delta t$. The corresponding coarse grid problem,

$$A_\Delta = \begin{pmatrix} I & & & \\ -\Phi_\Delta & I & & \\ & \ddots & \ddots & \\ & & -\Phi_\Delta & I \end{pmatrix},$$

is obtained by defining coarse grid propagators $\{\Phi_\Delta\}$ which are at least as cheap to apply as the fine scale propagators $\{\Phi\}$. The matrix A_Δ has fewer rows and columns than A, e.g., if we are coarsening in time by 2, A_Δ has one half as many rows and columns.

This coarse time grid induces a partition of the fine grid into C-points (associated with coarse grid points) and F-points, as visualized next. C-points exist on both the fine and coarse time grid, but F-points exist only on the fine time scale.



Every multigrid algorithm requires a relaxation method and an approach to transfer values between grids. Our relaxation scheme alternates between so-called F-relaxation and C-relaxation as illustrated next. F-relaxation updates the F-point values $\{u_j\}$ on interval (T_i, T_{i+1}) by simply propagating the C-point value u_{mi} across the interval using the time propagator $\{\Phi\}$. While this is a sequential process, each F-point interval update is independent from the others and can be computed in parallel. Similarly, C-relaxation updates the C-point value u_{mi} based on the F-point value u_{mi-1} and these updates can also be computed in parallel. This approach to relaxation can be thought of as line relaxation in space in that the residual is set to 0 for an entire time step.

The F updates are done simultaneously in parallel, as depicted next.

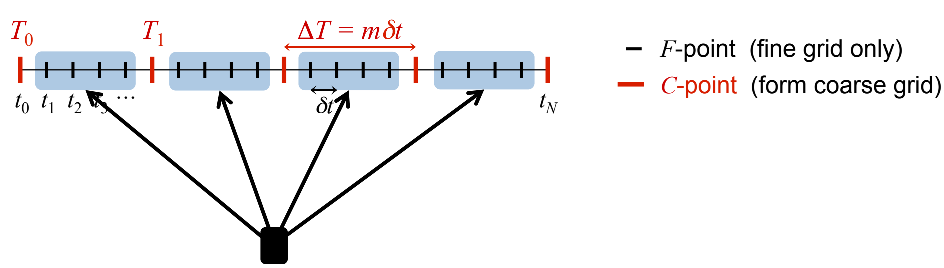


Figure 3: Update all F-point intervals in parallel, using the time propagator Φ.

Following the F sweep, the C updates are also done simultaneously in parallel, as depicted next.

[1] Ries, Manfred, Ulrich Trottenberg, and Gerd Winter. "A note on MGR methods." Linear Algebra and its Applications 49 (1983): 1-26.

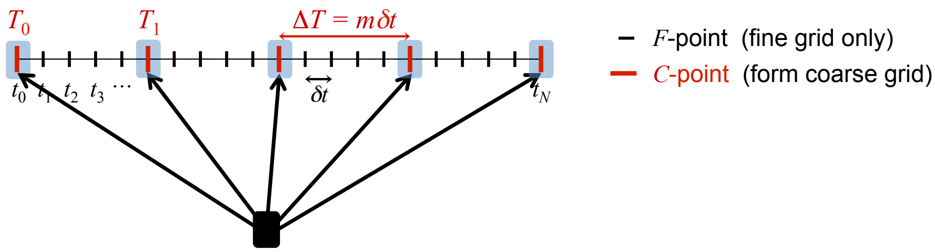


Figure 4: Update all C-points in parallel, using the time propagator Φ.

In general, FCF- and F-relaxation will refer to the relaxation methods used in XBraid. We can say

- FCF- or F-relaxation is highly parallel.
- But, a sequential component exists equaling the number of F-points between two C-points.
- XBraid uses regular coarsening factors, i.e., the spacing of C-points happens every m points.

After relaxation, comes forming the coarse grid error correction. To move quantities to the coarse grid, we use the restriction operator R which simply injects values at C-points from the fine grid to the coarse grid,

$$R = \begin{pmatrix} I & & \\ 0 & & \\ \vdots & & \\ 0 & & \\ & I & \\ & 0 & \\ & \vdots & \\ & 0 & \\ & & \ddots \end{pmatrix}^T .$$

The spacing between each I is $m-1$ block rows. While injection is simple, XBraid always does an F-relaxation sweep before the application of R, which is equivalent to using the transpose of harmonic interpolation for restriction (see `Parallel Time Integration with Multigrid`).

To define the coarse grid equations, we apply the Full Approximation Scheme (FAS) method, which is a nonlinear version of multigrid. This is to accommodate the general case where f is a nonlinear function. In FAS, the solution guess and residual (i.e., $\mathbf{u}, \mathbf{g} - A\mathbf{u}$) are restricted. This is in contrast to linear multigrid which typically restricts only the residual equation to the coarse grid. This algorithmic change allows for the solution of general nonlinear problems. For more details, see `PDF` by Van Henson for a good introduction to FAS. However, FAS was originally invented by Achi Brandt.

A central question in applying FAS is how to form the coarse grid matrix A_Δ, which in turn asks how to define the coarse grid time stepper Φ_Δ. One of the simplest choices (and one frequently used in practice) is to let Φ_Δ simply be Φ but with the coarse time step size $\Delta T = m\delta t$. For example, if $\Phi = (I - \delta t A)^{-1}$ for some backward Euler scheme, then $\Phi_\Delta = (I - m\delta t A)^{-1}$ would be one choice.

With a Φ_Δ defined, the coarse grid equation

$$A_\Delta(\mathbf{v}_\Delta) = A_\Delta(\mathbf{u}_\Delta) + \mathbf{r}_\Delta$$

is then solved. Finally, FAS defines a coarse grid error approximation $\mathbf{e}_\Delta = \mathbf{v}_\Delta - \mathbf{u}_\Delta$, which is interpolated with P_Φ back to the fine grid and added to the current solution guess. Interpolation is equivalent to injecting the coarse grid to the C-points on the fine grid, followed by an F-relaxation sweep (i.e., it is equivalent to harmonic interpolation, as mentioned above about restriction). That is,

$$P_\Phi = \begin{pmatrix} I & & \\ \Phi & & \\ \Phi^2 & & \\ \vdots & & \\ \Phi^{m-1} & & \\ & I & \\ & \Phi & \\ & \Phi^2 & \\ & \vdots & \\ & \Phi^{m-1} & \\ & & \ddots \end{pmatrix},$$

where m is the coarsening factor. See Two-Grid Algorithm for a concise description of the FAS algorithm for MGRIT.

2.3.1 Two-Grid Algorithm

The two-grid FAS process is captured with this algorithm. Using a recursive coarse grid solve (i.e., step 3 becomes a recursive call) makes the process multilevel. Halting is done based on a residual tolerance. If the operator is linear, this FAS cycle is equivalent to standard linear multigrid. Note that we represent A as a function below, whereas the above notation was simplified for the linear case.

1. Relax on $A(\mathbf{u}) = \mathbf{g}$ using FCF-relaxation
2. Restrict the fine grid approximation and its residual:

   $$\mathbf{u}_\Delta \leftarrow R\mathbf{u}, \quad \mathbf{r}_\Delta \leftarrow R(\mathbf{g} - A(\mathbf{u}),$$

   which is equivalent to updating each individual time step according to

   $$u_{\Delta,i} \leftarrow u_{mi}, \quad r_{\Delta,i} \leftarrow g_{mi} - A(\mathbf{u})_{mi} \quad \text{for} \quad i = 0, ..., N_\Delta.$$

3. Solve $A_\Delta(\mathbf{v}_\Delta) = A_\Delta(\mathbf{u}_\Delta) + \mathbf{r}_\Delta$
4. Compute the coarse grid error approximation: $\mathbf{e}_\Delta = \mathbf{v}_\Delta - \mathbf{u}_\Delta$
5. Correct: $\mathbf{u} \leftarrow \mathbf{u} + P\mathbf{e}_\Delta$
   This is equivalent to updating each individual time step by adding the error to the values of $\mathbf{u}$ at the C-points:

   $$u_{mi} = u_{mi} + e_{\Delta,i},$$

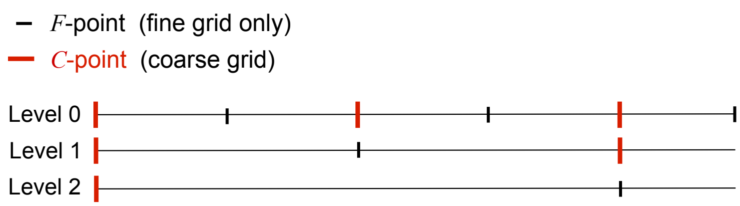
   followed by an F-relaxation sweep applied to $\mathbf{u}$.

2.3.2 Summary

In summary, a few points are

- XBraid is an iterative solver for the global space-time problem.
- The user defines the time stepping routine Φ and can wrap existing code to accomplish this.
- XBraid convergence will depend heavily on how well Φ_Δ approximates Φ^m, that is how well a time step size of $m\delta t = \Delta T$ will approximate m applications of the same time integrator for a time step size of δt. This is a subject of research, but this approximation need not capture fine scale behavior, which is instead captured by relaxation on the fine grid.

- The coarsest grid is solved exactly, i.e., sequentially, which can be a bottleneck for two-level methods like Parareal, [2] but not for a multilevel scheme like XBraid where the coarsest grid is of trivial size.
- By forming the coarse grid to have the same sparsity structure and time stepper as the fine grid, the algorithm can recur easily and efficiently.
- Interpolation is ideal or exact, in that an application of interpolation leaves a zero residual at all F-points.
- The process is applied recursively until a trivially sized temporal grid is reached, e.g., 2 or 3 time points. Thus, the coarsening rate m determines how many levels there are in the hierarchy. For instance in this figure, a 3 level hierarchy is shown. Three levels are chosen because there are six time points, $m = 2$ and $m^2 < 6 \leq m^3$. If the coarsening rate had been $m = 4$ then there would only be two levels because, there would be no more points to coarsen!



By default, XBraid will subdivide the time domain into evenly sized time steps. XBraid is structured to handle variable time step sizes and adaptive time step sizes, and these features are coming.

2.4 Overview of the XBraid Code

XBraid is designed to run in conjunction with an existing application code that can be wrapped per our interface. This application code will implement some time marching simulation like fluid flow. Essentially, the user has to take their application code and extract a stand-alone time-stepping function Φ that can evolve a solution from one time value to another, regardless of time step size. After this is done, the XBraid code takes care of the parallelism in the time dimension.

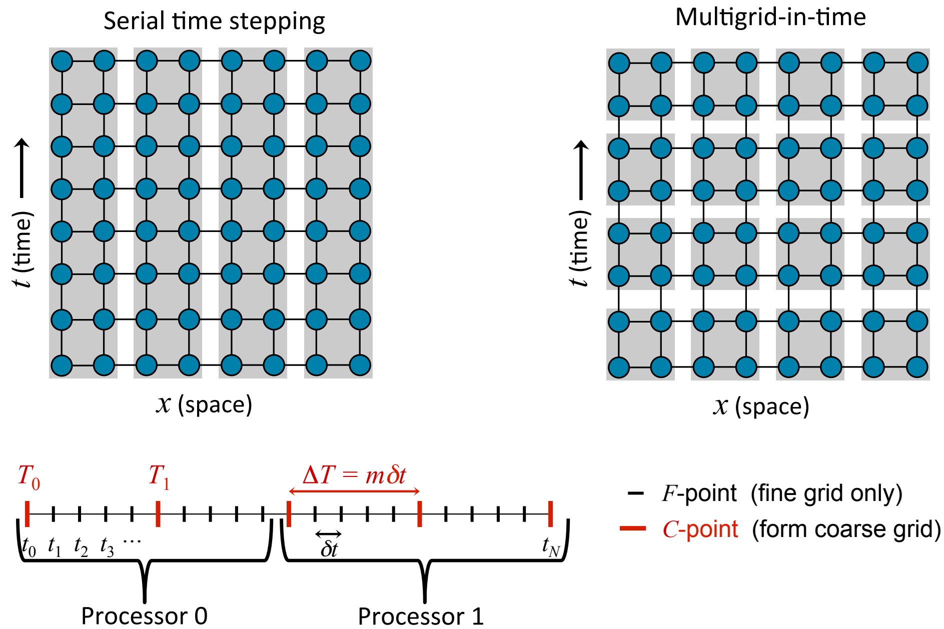
XBraid

- is written in C and can easily interface with Fortran and C++
- uses MPI for parallelism
- self documents through comments in the source code and through $*$.md files
- functions and structures are prefixed by *braid*
  - User routines are prefixed by `braid_`
  - Developer routines are prefixed by `_braid_`

2.4.1 Parallel decomposition and memory

- XBraid decomposes the problem in parallel as depicted next. As you can see, traditional time stepping only stores one time step at a time, but only enjoys a spatial data decomposition and spatial parallelism. On the other hand, XBraid stores multiple time steps simultaneously and each processor holds a space-time chunk reflecting both the spatial and temporal parallelism.
- XBraid only handles temporal parallelism and is agnostic to the spatial decomposition. See braid_SplitCommworld. Each processor owns a certain number of CF intervals of points. In the following figure, processor 1 and processor 2 each own 2 CF intervals. XBraid distributes intervals evenly on the finest grid.

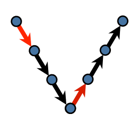
[2] Lions, J., Yvon Maday, and Gabriel Turinici. "A"parareal"in time discretization of PDE's." Comptes Rendus de l'Academie des Sciences Series I Mathematics 332.7 (2001): 661-668.



- XBraid increases the parallelism significantly, but now several time steps need to be stored, requiring more memory. XBraid employs two strategies to address the increased memory costs.
  - First, one need not solve the whole problem at once. Storing only one space-time slab is advisable. That is, solve for as many time steps (say k time steps) as you have available memory for. Then move on to the next k time steps.
  - Second, XBraid only stores the C-points. Whenever an F-point is needed, it is generated by F-relaxation. More precisely, we only store the red C-point time values in the previous figure. Coarsening is usually aggressive with $m = 8, 16, 32, \ldots$, so the storage requirements of XBraid are significantly reduced when compared to storing all of the time values.

2.4.2 Cycling and relaxation strategies

There are two main cycling strategies available in XBraid, F-and V-cycles. These two cycles differ in how often and the order in which coarse levels are visited. A V-cycle is depicted next, and is a simple recursive application of the Two-Grid Algorithm.

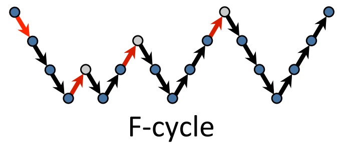


V-cycle

An F-cycle visits coarse grids more frequently and in a different order. Essentially, an F-cycle uses a V-cycle as the post-smoother, which is an expensive choice for relaxation. But, this extra work gives you a closer approximation to a two-grid cycle, and a faster convergence rate at the extra expense of more work. The effectiveness of a V-cycle as a relaxation scheme can be seen in Figure 2, where one V-cycle globally propagates and *smoothes* the error. The cycling strategy of an F-cycle is depicted next.

Next, we make a few points about F- versus V-cycles.

- One V-cycle iteration is cheaper than one F-cycle iteration.



- But, F-cycles often converge more quickly. For some test cases, this difference can be quite large. The cycle choice for the best time to solution will be problem dependent. See Scaling Study with this Example for a case study of cycling strategies.

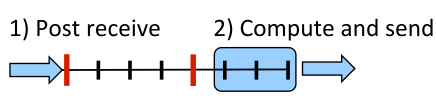
The number of FC relaxation sweeps is another important algorithmic setting. Note that at least one F-relaxation sweep is always done on a level. A few summary points about relaxation are as follows.

- Using FCF (or even FCFCF, FCFCFCF) relaxation, corresponding to passing *braid_SetNRelax* a value of 1, 2 or 3 respectively, will result in an XBraid cycle that converges more quickly as the number of relaxations grows.
- But as the number of relaxations grows, each XBraid cycle becomes more expensive. The optimal relaxation strategy for the best time to solution will be problem dependent.
- However, a good first step is to try FCF on all levels (i.e., *braid_SetNRelax(core, -1, 1)*).
- A common optimization is to first set FCF on all levels (i.e., *braid_setnrelax(core, -1, 1)*), but then overwrite the FCF option on level 0 so that only F-relaxation is done on level 0, (i.e., *braid_setnrelax(core, 0, 1)*). This strategy can work well with F-cycles.
- See Scaling Study with this Example for a case study of relaxation strategies.

Last, `Parallel Time Integration with Multigrid` has a more in depth case study of cycling and relaxation strategies

2.4.3 Overlapping communication and computation

XBraid effectively overlaps communication and computation. The main computational kernel of XBraid is one relaxation sweep touching all the CF intervals. At the start of a relaxation sweep, each process first posts a non-blocking receive at its left-most point. It then carries out F-relaxation in each interval, starting with the right-most interval to send the data to the neighboring process as soon as possible. If each process has multiple CF intervals at this XBraid level, the strategy allows for complete overlap.



2.4.4 Configuring the XBraid Hierarchy

Some of the more basic XBraid function calls allow you to control aspects discussed here.

- braid_SetFMG: switches between using F- and V-cycles.
- braid_SetMaxIter: sets the maximum number of XBraid iterations
- braid_SetCFactor: sets the coarsening factor for any (or all levels)
- braid_SetNRelax: sets the number of CF-relaxation sweeps for any (or all levels)
- braid_SetRelTol, braid_SetAbsTol: sets the stopping tolerance
- braid_SetMinCoarse: sets the minimum possible coarse grid size

- braid_SetMaxLevels: sets the maximum number of levels in the XBraid hierarchy

2.4.5 Halting tolerance

Another important configuration aspect regards setting a residual halting tolerance. Setting a tolerance involves these three XBraid options:

1. braid_PtFcnSpatialNorm
   This user-defined function carries out a spatial norm by taking the norm of a braid_Vector. A common choice is the standard Eucliden norm (2-norm), but many other choices are possible, such as an L2-norm based on a finite element space.
2. braid_SetTemporalNorm
   This option determines how to obtain a global space-time residual norm. That is, this decides how to combine the spatial norms returned by braid_PtFcnSpatialNorm at each time step to obtain a global norm over space and time. It is this global norm that then controls halting.
   There are three options for setting the *tnorm* value passed to braid_SetTemporalNorm. We let the summation index *i* be over all C-point values on the fine time grid, *k* refer to the current XBraid iteration, *r* be residual values, *space_time* norms be a norm over the entire space-time domain and *spatial_norm* be the user-defined spatial norm from braid_PtFcnSpatialNorm. Thus, r_i is the residual at the *ith* C-point, and $r^{(k)}$ is the residual at the *kth* XBraid iteration. The three options are then defined as,
   - *tnorm=1*: One-norm summation of spatial norms
     $$\|r^{(k)}\|_{\text{space\_time}} = \Sigma_i \|r_i^{(k)}\|_{\text{spatial\_norm}}$$
     If braid_PtFcnSpatialNorm is the one-norm over space, then this is equivalent to the one-norm of the global space-time residual vector.
   - *tnorm=2*: Two-norm summation of spatial norms
     $$\|r^{(k)}\|_{\text{space\_time}} = \left(\Sigma_i \|r_i^{(k)}\|^2_{\text{spatial\_norm}}\right)^{1/2}$$
     If braid_PtFcnSpatialNorm is the Euclidean norm (two-norm) over space, then this is equivalent to the Euclidean-norm of the global space-time residual vector.
   - *tnorm=3*: Infinity-norm combination of spatial norms
     $$\|r^{(k)}\|_{\text{space\_time}} = \max_i \|r_i^{(k)}\|_{\text{spatial\_norm}}$$
     If braid_PtFcnSpatialNorm is the infinity-norm over space, then this is equivalent to the infinity-norm of the global space-time residual vector.

   The default choice is *tnorm=2*
3. braid_SetAbsTol, braid_SetRelTol
   - If an absolute tolerance is used, then
     $$\|r^{(k)}\|_{\text{space\_time}} < \text{tol}$$
     defines when to halt.
   - If a relative tolerance is used, then
     $$\frac{\|r^{(k)}\|_{\text{space\_time}}}{\|r^{(0)}\|_{\text{space\_time}}} < \text{tol}$$
     defines when to halt. That is, the current *kth* residual is scaled by the initial residual before comparison to the halting tolerance. This is similar to typical relative residual halting tolerances used in spatial multigrid, but can be a dangerous choice in this setting.

Care should be practiced when choosing a halting tolerance. For instance, if a relative tolerance is used, then issues can arise when the initial guess is zero for large numbers of time steps. Taking the case where the initial guess (defined by braid_PtFcnInit) is 0 for all time values $t > 0$, the initial residual norm will essentially only be nonzero at the first time value,

$$\|r^{(0)}\|_{\text{space\_time}} \approx \|r_1^{(k)}\|_{\text{spatial\_norm}}$$

This will skew the relative halting tolerance, especially if the number of time steps increases, but the initial residual norm does not.

A better strategy is to choose an absolute tolerance that takes your space-time domain size into account, as in Section Scaling Study with this Example, or to use an infinity-norm temporal norm option.

2.5 Citing XBraid

To cite XBraid, please state in your text the version number from the VERSION file, and please cite the project website in your bibliography as

[1] XBraid: Parallel multigrid in time. http://llnl.gov/casc/xbraid.

The corresponding BibTex entry is

```
@misc{xbraid-package,
  title = {{XB}raid: Parallel multigrid in time},
  howpublished = {\url{http://llnl.gov/casc/xbraid}}
  }
```

2.6 Summary

- XBraid applies multigrid to the time dimension.
  - This exposes concurrency in the time dimension.
  - The potential for speedup is large, 10x, 100x, ...
- This is a non-intrusive approach, with an unchanged time discretization defined by user.
- Parallel time integration is only useful beyond some scale. This is evidenced by the experimental results below. For smaller numbers of cores sequential time stepping is faster, but at larger core counts XBraid is much faster.
- The more time steps that you can parallelize over, the better your speedup will be.
- XBraid is optimal for a variety of parabolic problems (see the examples directory).

3 Examples

3.1 The Simplest Example

User Defined Structures and Wrappers

The user must wrap their existing time stepping routine per the XBraid interface. To do this, the user must define two data structures and some wrapper routines. To make the idea more concrete, we now give these function definitions from `examples/ex-01`, which implements a scalar ODE,

$$u_t = \lambda u.$$

The two data structures are:

1. **App**: This holds a wide variety of information and is *global* in that it is passed to every function. This structure holds everything that the user will need to carry out a simulation. Here, this is just the global MPI communicator and few values describing the temporal domain.
```
typedef struct _braid_App_struct
{
   MPI_Comm  comm;
   double    tstart;
   double    tstop;
   int       ntime;

} my_App;
```
2. **Vector**: this defines (roughly) a state vector at a certain time value. It could also contain any other information related to this vector which is needed to evolve the vector to the next time value, like mesh information. Here, the vector is just a scalar double.
```
typedef struct _braid_Vector_struct
{
   double value;

} my_Vector;
```

The user must also define a few wrapper routines. Note, that the *app* structure is the first argument to every function.

1. **Phi**: This function tells XBraid how to take a time step, and is the core user routine. The user must advance the vector *u* from time *tstart* to time *tstop*. Note how the time values are given to the user through the *status* structure and associated *Get* routines. The *rfactor_ptr* parameter is an advanced topic not used here.
Here advancing the solution just involves the scalar λ.
Importantly, the g_i function (from Overview of the XBraid Algorithm) must be incorporated into *Phi*, so that $\Phi(u_i) \rightarrow u_{i+1}$
```
int
my_Phi(braid_App       app,
       braid_Vector    u,
       braid_PhiStatus status)
{
   double tstart;             /* current time */
   double tstop;              /* evolve to this time*/
   braid_PhiStatusGetTstartTstop(status, &tstart, &tstop);

   /* On the finest grid, each value is half the previous value */
   (u->value) = pow(0.5, tstop-tstart)*(u->value);

   /* Zero rhs for now */
   (u->value) += 0.0;

   /* no refinement */
   braid_PhiStatusSetRFactor(status, 1);

   return 0;
}
```
2. **Init**: This function tells XBraid how to initialize a vector at time *t*. Here that is just allocating and setting a scalar on the heap.
```
int
my_Init(braid_App     app,
        double        t,
        braid_Vector *u_ptr)
{
   my_Vector *u;

   u = (my_Vector *) malloc(sizeof(my_Vector));
   if (t == 0.0)
   {
      /* Initial guess */
      (u->value) = 1.0;
   }
```

```
      else
      {
         /* Random between 0 and 1 */
         (u->value) = ((double)rand()) / RAND_MAX;
      }
      *u_ptr = u;

      return 0;
   }
```

3. **Clone**: This function tells XBraid how to clone a vector into a new vector.

```
   int
   my_Clone(braid_App     app,
            braid_Vector  u,
            braid_Vector *v_ptr)
   {
      my_Vector *v;

      v = (my_Vector *) malloc(sizeof(my_Vector));
      (v->value) = (u->value);
      *v_ptr = v;

      return 0;
   }
```

4. **Free**: This function tells XBraid how to free a vector.

```
   int
   my_Free(braid_App    app,
           braid_Vector u)
   {
      free(u);

      return 0;
   }
```

5. **Sum**: This function tells XBraid how to sum two vectors (AXPY operation).

```
   int
   my_Sum(braid_App    app,
          double    alpha,
          braid_Vector x,
          double    beta,
          braid_Vector y)
   {
      (y->value) = alpha*(x->value) + beta*(y->value);

      return 0;
   }
```

6. **SpatialNorm**: This function tells XBraid how to take the norm of a *braid_Vector* and is used for halting. This norm is only over space. A common norm choice is the standard Euclidean norm, but many other choices are possible, such as an L2-norm based on a finite element space. The norm choice should be based on what makes sense for you problem. How to accumulate spatial norm values to obtain a global space-time residual norm for halting decisions is controlled by braid_SetTemporalNorm.

```
   int
   my_SpatialNorm(braid_App     app,
                  braid_Vector  u,
                  double       *norm_ptr)
   {
      double dot;

      dot = (u->value)*(u->value);
      *norm_ptr = sqrt(dot);

      return 0;
   }
```

7. **Access**: This function allows the user access to XBraid and the current solution vector at time *t*. This is most commonly used to print solution(s) to screen, file, etc... The user defines what is appropriate output. Notice how you are told the time value *t* of the vector *u* and even more information in *astatus*. This lets you tailor the output

to only certain time values at certain XBraid iterations. Querying *astatus* for such information is done through *braid_AccessStatusGet**(..)* routines.

The frequency of the calls to *access* is controlled through braid_SetAccessLevel. For instance, if *access_level* is set to 2, then *access* is called every XBraid iteration and on every XBraid level. In this case, querying *astatus* to determine the current XBraid level and iteration will be useful. This scenario allows for even more detailed tracking of the simulation.

Eventually, this routine will allow for broader access to XBraid and computational steering.

See examples/ex-02 and drivers/drive-04 for more advanced uses of the *access* function. Drive-04 uses *access* to write solution vectors to a GLVIS visualization port, and examples/ex-02 uses *access* to write to .vtu files.

```
int
my_Access(braid_App          app,
          braid_Vector       u,
          braid_AccessStatus astatus)
{
   MPI_Comm   comm   = (app->comm);
   double     tstart = (app->tstart);
   double     tstop  = (app->tstop);
   int        ntime  = (app->ntime);
   int        index, myid;
   char       filename[255];
   FILE      *file;
   double     t;

   braid_AccessStatusGetT(astatus, &t);
   index = ((t-tstart) / ((tstop-tstart)/ntime) + 0.1);

   MPI_Comm_rank(comm, &myid);

   sprintf(filename, "%s.%07d.%05d", "ex-01.out", index, myid);
   file = fopen(filename, "w");
   fprintf(file, "%.14e\n", (u->value));
   fflush(file);
   fclose(file);

   return 0;
}
```

8. **BufSize**, **BufPack**, **BufUnpack**: These three routines tell XBraid how to communicate vectors between processors. *BufPack* packs a vector into a `void *` buffer for MPI and then *BufUnPack* unpacks it from `void *` to vector. Here doing that for a scalar is trivial. *BufSize* computes the upper bound for the size of an arbitrary vector.

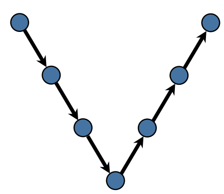
   Note how *BufPack* also returns a size pointer. This size pointer should be the exact number of bytes packed, while *BufSize* should provide only an upper-bound on a possible buffer size. This flexibility allows for variable spatial grid sizes to result in smaller messages sent when appropriate. **To avoid MPI issues, it is very important that BufSize be pessimistic, provide an upper bound, and return the same value across processors.**

   In general, the buffer should be self-contained. The receiving processor should be able to pull all necessary information from the buffer in order to properly interpret and unpack the buffer.

```
int
my_BufSize(braid_App  app,
           int       *size_ptr)
{
   *size_ptr = sizeof(double);
   return 0;
}

int
my_BufPack(braid_App     app,
           braid_Vector  u,
```

```
                  void         *buffer,
                  braid_Int    *size_ptr)
{
   double *dbuffer = buffer;

   dbuffer[0] = (u->value);
   *size_ptr = sizeof(double);

   return 0;
}

int
my_BufUnpack(braid_App     app,
             void         *buffer,
             braid_Vector *u_ptr)
{
   double    *dbuffer = buffer;
   my_Vector *u;

   u = (my_Vector *) malloc(sizeof(my_Vector));
   (u->value) = dbuffer[0];
   *u_ptr = u;

   return 0;
}
```

9. **Coarsen**, **Restrict** (optional): These are advanced options that allow for coarsening in space while you coarsen in time. This is useful for maintaining stable explicit schemes on coarse time scales and is not needed here. See for instance `drivers/drive-04` and `drivers/drive-02` which use these routines.
   These functions allow you vary the spatial mesh size on XBraid levels as depicted here where the spatial and temporal grid sizes are halved every level.



10. Adaptive and variable time stepping is in the works to be implemented. The *rfactor* parameter in *Phi* will allow this.

Running XBraid for this Example

A typical flow of events in the *main* function is to first initialize the *app* structure.

```
/* set up app structure */
app = (my_App *) malloc(sizeof(my_App));
(app->comm)   = comm;
(app->tstart) = tstart;
(app->tstop)  = tstop;
(app->ntime)  = ntime;
```

Then, the data structure definitions and wrapper routines are passed to XBraid. The core structure is used by XBraid for internal data structures.

```
braid_Core  core;
braid_Init(MPI_COMM_WORLD, comm, tstart, tstop, ntime, app,
        my_Phi, my_Init, my_Clone, my_Free, my_Sum, my_SpatialNorm,
        my_Access, my_BufSize, my_BufPack, my_BufUnpack, &core);
```

Then, XBraid options are set.

```
braid_SetPrintLevel( core, 1);
braid_SetMaxLevels(core, max_levels);
braid_SetNRelax(core, -1, nrelax);
braid_SetAbsTol(core, tol);
braid_SetCFactor(core, -1, cfactor);
braid_SetMaxIter(core, max_iter);
```

Then, the simulation is run.

```
braid_Drive(core);
```

Then, we clean up.

```
braid_Destroy(core);
```

Finally, to run ex-01, type

```
ex-01 -ml 5
```

This will run `ex-01`. See `examples/ex-0*` for more extensive examples.

3.2 Two-Dimensional Heat Equation

In this example, we assume familiarity with The Simplest Example and describe the major ways in which this example differs. This example is a full space-time parallel example, as opposed to The Simplest Example, which implements only a scalar ode for one degree-of-freedom in space. We solve the heat equation in 2D,

$$\delta/\delta_t \; u(x,y,t) = \Delta u(x,y,t) + g(x,y,t).$$

For spatial parallelism, we rely on the `hypre` package where the SemiStruct interface is used to define our spatial discretization stencil and form our time stepping scheme, the backward Euler method. The spatial discretization is just the standard 5-point finite difference stencil ($[-1;-1,4,-1;-1]$), scaled by mesh widths, and the PFMG solver is used for the solves required by backward Euler. Please see the hypre manual and examples for more information on the SemiStruct interface and PFMG. Although, the hypre specific calls have mostly been abstracted away for this example, and so it is not necessary to be familiar with the SemiStruct interface for this example.

This example consists of three files and two executables.

- examples/ex-02-serial.c: This file compiles into its own executable `ex-02-serial` and represents a simple example user application. This file supports only parallelism in space and represents a basic approach to doing efficient sequential time stepping with the backward Euler scheme. Note that the hypre solver used (PFMG) to carry out the time stepping is highly efficient.
- examples/ex-02.c: This file compiles into its own executable `ex-02` and represents a basic example of wrapping the user application `ex-02-serial`. We will go over the wrappers below.
- ex-02-lib.c: This file contains shared functions used by the time-serial version and the time-parallel version. This is where most of the hypre specific calls reside. This file provides the basic functionality of this problem. For instance, *take_step(u, tstart, tstop, ...)* carries out a step, moving the vector *u* from time *tstart* to time *tstop* and *setUpImplicitMatrix(...)* constructs the matrix to be inverted by PFMG for the backward Euler method.

User Defined Structures and Wrappers

We now discuss in more detail the important data structures and wrapper routines in `examples/ex-02.c`. The actual code for this example is quite simple and it is recommended to read through it after this overview.

The two data structures are:

1. **App**: This holds a wide variety of information and is *global* in that it is passed to every user function. This structure holds everything that the user will need to carry out a simulation. One important structure contained in the *app* is the *simulation_manager*. This is a structure native to the user code `ex-02-lib.c`. This structure conveniently holds the information needed by the user code to carry out a time step. For instance,
   `app->man->A`
   is the time stepping matrix,
   `app->man->solver`
   is the hypre PFMG solver object,
   `app->man->dt`
   is the current time step size. The app is defined as

```
typedef struct _braid_App_struct {
   MPI_Comm             comm;             /* global communicator */
   MPI_Comm             comm_t;           /* communicator for parallelizing in time  */
   MPI_Comm             comm_x;           /* communicator for parallelizing in space  */
   int                  pt;               /* number of processors in time  */
   simulation_manager  *man;              /* user's simulation manager structure */
   HYPRE_SStructVector  e;                /* temporary vector used for error computations */
   int                  nA;               /* number of spatial matrices created */
   HYPRE_SStructMatrix *A;                /* array of spatial matrices, size nA, one per level*/
   double              *dt_A;             /* array of time step sizes, size nA, one per level*/
   HYPRE_StructSolver  *solver;           /* array of PFMG solvers, size nA, one per level*/
   int                  use_rand;         /* binary value, use random or zero initial guess */
   int                 *runtime_max_iter; /* runtime info for number of PFMG iterations*/
   int                 *max_iter_x;       /* maximum iteration limits for PFMG */
} my_App;
```

The app contains all the information needed to take a time step with the user code for an arbitrary time step size. See the *Phi* function below for more detail.

1. **Vector**: this defines a state vector at a certain time value. Here, the vector is a structure containing a native hypre data-type, the *SStructVector*, which describes a vector over the spatial grid. Note that *my_Vector* is used to define *braid_Vector*.

```
typedef struct _braid_Vector_struct {
   HYPRE_SStructVector   x;
} my_Vector;
```

The user must also define a few wrapper routines. Note, that the `app` structure is the first argument to every function.

1. **Phi**: This function tells XBraid how to take a time step, and is the core user routine. This function advances the vector *u* from time *tstart* to time *tstop*. A few important things to note are as follows.
   - The time values are given to the user through the *status* structure and associated *Get* routines.
   - The basic strategy is to see if a matrix and solver already exist for this dt value. If not, generate a new matrix and solver and store them in the *app* structure. If they do already exist, then re-use the data.
   - To carry out a step, the user routines from `ex-02-lib.c` rely on a few crucial data members *man->dt*, *man->A* and *man-solver*. We overwrite these members with the correct information for the time step size in question. Then, we pass *man* and *u* to the user function *take_step(...)* which evolves *u*.
   - The forcing term g_i is wrapped into the *take_step(...)* function. Thus, $\Phi(u_i) \to u_{i+1}$.

```
int my_Phi(braid_App       app,
           braid_Vector    u,
           braid_PhiStatus status)
{
   double tstart;             /* current time */
   double tstop;              /* evolve u to this time*/
   int i, A_idx;
   int iters_taken = -1;

   /* Grab status of current time step */
   braid_PhiStatusGetTstartTstop(status, &tstart, &tstop);
```

```
        /* Check matrix lookup table to see if this matrix already exists*/
        A_idx = -1.0;
        for( i = 0; i < app->nA; i++ ){
           if( fabs( app->dt_A[i] - (tstop-tstart) )/(tstop-tstart) < 1e-10) {
              A_idx = i;
              break;
           }
        }

        /* We need to "trick" the user's manager with the new dt */
        app->man->dt = tstop - tstart;

        /* Set up a new matrix and solver and store in app */
        if( A_idx == -1.0 ){
           A_idx = i;
           app->nA++;
           app->dt_A[A_idx] = tstop-tstart;

           setUpImplicitMatrix( app->man );
           app->A[A_idx] = app->man->A;

           setUpStructSolver( app->man, u->x, u->x );
           app->solver[A_idx] = app->man->solver;
        }

        /* Time integration to next time point: Solve the system Ax = b.
         * First, "trick" the user's manager with the right matrix and solver */
        app->man->A = app->A[A_idx];
        app->man->solver = app->solver[A_idx];
        ...
        /* Take step */
        take_step(app->man, u->x, tstart, tstop);
        ...
        return 0;
     }
```

2. There are other functions, **Init**, **Clone**, **Free**, **Sum**, **SpatialNorm**, **Access**, **BufSize**, **BufPack** and **BufUnpack**, which also must be written. These functions are all simple for this example, as for the case of The Simplest Example. All we do here is standard operations on a spatial vector such as initialize, clone, take an inner-product, pack, etc... We refer the reader to `ex-02.c`.

Running XBraid for this Example

To initialize and run XBraid, the procedure is similar to The Simplest Example. Only here, we have to both initialize the user code and XBraid. The code that is specific to the user's application comes directly from the existing serial simulation code. If you compare `ex-02-serial.c` and `ex-02.c`, you will see that most of the code setting up the user's data structures and defining the wrapper functions are simply lifted from the serial simulation.

Taking excerpts from the function *main()* in ex-02.c, we first initialize the user's simulation manager with code like

```
...
app->man->px    = 1;   /* my processor number in the x-direction */
app->man->py    = 1;   /* my processor number in the y-direction */
                       /* px*py=num procs in space */
app->man->nx    = 17;  /* number of points in the x-dim */
app->man->ny    = 17;  /* number of points in the y-dim */
app->man->nt    = 32;  /* number of time steps */
...
```

We also define default XBraid parameters with code like

```
...
max_levels      = 15; /* Max levels for XBraid solver */
min_coarse      = 3;  /* Minimum possible coarse grid size */
```

```
  nrelax          = 1;  /* Number of CF relaxation sweeps on all levels */
  ...
```

The XBraid app must also be initialized with code like

```
...
app->comm   = comm;
app->tstart = tstart;
app->tstop  = tstop;
app->ntime  = ntime;
```

Then, the data structure definitions and wrapper routines are passed to XBraid.

```
braid_Core  core;
braid_Init(MPI_COMM_WORLD, comm, tstart, tstop, ntime, app,
        my_Phi, my_Init, my_Clone, my_Free, my_Sum, my_SpatialNorm,
        my_Access, my_BufSize, my_BufPack, my_BufUnpack, &core);
```

Then, XBraid options are set with calls like

```
...
braid_SetPrintLevel( core, 1);
braid_SetMaxLevels(core, max_levels);
braid_SetNRelax(core, -1, nrelax);
...
```

Then, the simulation is run.

```
braid_Drive(core);
```

Then, we clean up.

```
braid_Destroy(core);
```

Finally, to run ex-02, type

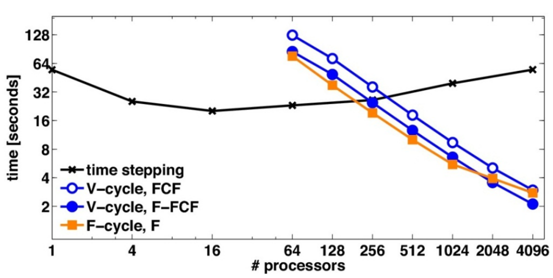
```
ex-02 -help
```

As a simple example, try the following.

```
mpirun -np 8 ex-02 -pgrid 2 2 2 -nt 256
```

3.2.1 Scaling Study with this Example

Here, we carry out a simple strong scaling study for this example. The "time stepping" data set represents sequential time stepping and was generated using `examples/ex-02-serial`. The time-parallel data set was generated using `examples/ex-02`. The problem setup is as follows.



- Backwards Euler is used as the time stepper. This is the only time stepper supported by `ex-02`.
- We used a Linux cluster with 4 cores per node, a Sandybridge Intel chipset, and a fast Infiniband interconnect.
- The space-time problem size was $129^2 \times 16,192$ over the unit cube $[0,1] \times [0,1] \times [0,1]$.
- The coarsening factor was $m = 16$ on the finest level and $m = 2$ on coarser levels.
- Since 16 processors optimized the serial time stepping approach, 16 processors in space are also used for the XBraid experiments. So for instance 512 processrs in the plot corresponds to 16 processors in space and 32 processors in time, $16 * 32 = 512$. Thus, each processor owns a space-time hypercube of $(129^2/16) \times (16,192/32)$. See Parallel decomposition and memory for a depiction of how XBraid breaks the problem up.
- Various relaxation and V and F cycling strategies are experimented with.
  - *V-cycle, FCF* denotes V-cycles and FCF-relaxation on each level.
  - *V-cycle, F-FCF* denotes V-cycles and F-relaxation on the finest level and FCF-relaxation on all coarser levels.
  - *F-cycle, F* denotes F-cycles and F-relaxation on each level.
- The initial guess at time values for $t > 0$ is zero, which is typical.
- The halting tolerance corresponds to a discrete L2-norm and was

  $$\text{tol} = \frac{10^{-8}}{\sqrt{(h_x)^2 h_t}},$$

  where h_x and h_t are the spatial and temporal grid spacings, respectively.
  This corresponds to passing *tol* to braid_SetAbsTol, passing *2* to braid_SetTemporalNorm and defining braid_PtFcnSpatialNorm to be the standard Euclidean 2-norm. All together, this appropriately scales the space-time residual in way that is relative to the number of space-time grid points (i.e., it approximates the L2-norm).

To re-run this scaling study, a sample run string for ex-02 is

```
mpirun -np 64 ex-02 -pgrid 4 4 4 -nx 129 129 -nt 16129 -cf0 16 -cf 2 -nu 1 -use_rand 0
```

To re-run the baseline sequential time stepper, ex-02-serial, try

```
mpirun -np 64 ex-02-serial -pgrid 8 8 -nx 129 129 -nt 16129
```

For explanations of the command line parameters, type

```
ex-02-serial -help
ex-02 -help
```

Regarding the performance, we can say

- The best speedup is 10x and this would grow if more processors were available.
- Although not shown, the iteration counts here are about 10-15 XBraid iterations. See `Parallel Time Integration with Multigrid` for the exact iteration counts.
- At smaller core counts, serial time stepping is faster. But at about 256 processors, there is a crossover and XBraid is faster.
- You can see the impact of the cycling and relaxation strategies discussed in Cycling and relaxation strategies. For instance, even though *V-cycle, F-FCF* is a weaker relaxation strategy than *V-cycle, FCF* (i.e., the XBraid convergence is slower), *V-cycle, F-FCF* has a faster time to solution than *V-cycle, FCF* because each cycle is cheaper.
- In general, one level of aggressive coarsening (here by a factor 16) followed by slower coarsening was found to be best on this machine.

Achieving the best speedup can require some tuning, and it is recommended to read `Parallel Time Integration with Multigrid` where this 2D heat equation example is explored in much more detail.

Running and Testing XBraid

The best overall test for XBraid, is to set the maximum number of levels to 1 (see braid_SetMaxLevels) which will carry out a sequential time stepping test. Take the output given to you by your *Access* function and compare it to output from a non-XBraid run. Is everything OK? Once this is complete, repeat for multilevel XBraid, and check that the solution is correct (that is, it matches a serial run to within tolerance).

At a lower level, to do sanity checks of your data structures and wrapper routines, there are also XBraid test functions, which can be easily run. The test routines also take as arguments the *app* structure, spatial communicator *comm_x*, a stream like *stdout* for test output and a time step size *dt* to test. After these arguments, function pointers to wrapper routines are the rest of the arguments. Some of the tests can return a boolean variable to indicate correctness.

```
/* Test init(), access(), free() */
braid_TestInitAccess( app, comm_x, stdout, dt, my_Init, my_Access, my_Free);

/* Test clone() */
braid_TestClone( app, comm_x, stdout, dt, my_Init, my_Access, my_Free, my_Clone);

/* Test sum() */
braid_TestSum( app, comm_x, stdout, dt, my_Init, my_Access, my_Free, my_Clone, my_Sum);

/* Test spatialnorm() */
correct = braid_TestSpatialNorm( app, comm_x, stdout, dt, my_Init, my_Free, my_Clone,
                          my_Sum, my_SpatialNorm);

/* Test bufsize(), bufpack(), bufunpack() */
correct = braid_TestBuf( app, comm_x, stdout, dt, my_Init, my_Free, my_Sum, my_SpatialNorm,
                    my_BufSize, my_BufPack, my_BufUnpack);

/* Test coarsen and refine */
correct = braid_TestCoarsenRefine(app, comm_x, stdout, 0.0, dt, 2*dt, my_Init,
                       my_Access, my_Free, my_Clone, my_Sum, my_SpatialNorm,
                       my_CoarsenInjection, my_Refine);
correct = braid_TestCoarsenRefine(app, comm_x, stdout, 0.0, dt, 2*dt, my_Init,
                      my_Access, my_Free, my_Clone, my_Sum, my_SpatialNorm,
                      my_CoarsenBilinear, my_Refine);
```

4 Building XBraid

- Copyright information and licensing restrictions can be found in the files COPYRIGHT and LICENSE.
- To specify the compilers, flags and options for your machine, edit makefile.inc. For now, we keep it simple and avoid using configure or cmake.
- To make the library, libbraid.a,
  `$ make`
- To make the examples
  `$ make all`
- The makefile lets you pass some parameters like debug with
  `$ make debug=yes`
  or
  `$ make all debug=yes`
  It would also be easy to add additional parameters, e.g., to compile with insure.
- To set compilers and library locations, look in makefile.inc where you can set up an option for your machine to define simple stuff like
  ```
  CC = mpicc
  MPICC = mpicc
  ```

```
MPICXX = mpiCC
LFLAGS = -lm
```

5 Examples: compiling and running

Type

```
ex-0* -help
```

for instructions on how to run any example.

To run the examples, type

```
mpirun -np 4 ex-*  [args]
```

1. ex-01 is the simplest example. It implements a scalar ODE and can be compiled and run with no outside dependencies. See Section (The Simplest Example) for more discussion of this example.
2. ex-02 implements the 2D heat equation on a regular grid. You must have hypre installed and these variables in examples/Makefile set correctly
   ```
   HYPRE_DIR = ../../linear_solvers/hypre
   HYPRE_FLAGS = -I$(HYPRE_DIR)/include
   HYPRE_LIB = -L$(HYPRE_DIR)/lib -lHYPRE
   ```
   Only implicit time stepping (backward Euler) is supported. See Section (Two-Dimensional Heat Equation) for more discussion of this example. The driver
   ```
   drivers/drive-02
   ```
   is a more sophisticated version of this simple example that supports explicit time stepping and spatial coarsening.

6 Drivers: compiling and running

Type

```
drive-0* -help
```

for instructions on how to run any driver.

To run the examples, type

```
mpirun -np 4 drive-*  [args]
```

1. drive-02 implements the 2D heat equation on a regular grid. You must have hypre installed and these variables in examples/Makefile set correctly
   ```
   HYPRE_DIR = ../../linear_solvers/hypre
   HYPRE_FLAGS = -I$(HYPRE_DIR)/include
   HYPRE_LIB = -L$(HYPRE_DIR)/lib -lHYPRE
   ```
   This driver also support spatial coarsening and explicit time stepping. This allows you to use explicit time stepping on each Braid level, regardless of time step size.
2. drive-03 implements the 3D heat equation on a regular grid, and assumes hypre is installed just like drive-02. This driver does not support spatial coarsening, and thus if explicit time stepping is used, the time stepping switchs to implicit on coarse XBraid grids when the CFL condition is violated.
3. drive-04 is a sophisticated test bed for various PDEs, mostly parabolic. It relies on the mfem package to create general finite element discretizations for the spatial problem. Other packages must be installed in this order.
   - Unpack and install Metis
   - Unpack and install hypre

- Unpack and install mfem. Make the serial version of mfem first by only typing `make`. Then make sure to set these variables correctly in the mfem Makefile:
```
USE_METIS_5 = YES
HYPRE_DIR  = where_ever_linear_solvers_is/hypre
```
- Make GLVIS, which needs serial mfem. Set these variables in the glvis makefile
```
MFEM_DIR   = mfem_location
MFEM_LIB   = -L$(MFEM_DIR) -lmfem
```
- Go back to the mfem directory and type
```
make clean
make parallel
```
- Go to braid/examples and set these Makefile variables,
```
METIS_DIR = ../../metis-5.1.0/lib
MFEM_DIR = ../../mfem
MFEM_FLAGS = -I$(MFEM_DIR)
MFEM_LIB = -L$(MFEM_DIR) -lmfem -L$(METIS_DIR) -lmetis
```
then type
```
make drive-04
```
- To run drive-04 and glvis, open two windows. In one, start a glvis session
```
./glvis
```
Then, in the other window, run drive-04
```
mpirun -np ... drive-04 [args]
```
Glvis will listen on a port to which drive-04 will dump visualization information.

7 Module Index

7.1 Modules

Here is a list of all modules:

8 File Index

8.1 File List

Here is a list of all files with brief descriptions:

9 Module Documentation

9.1 User-written routines

Typedefs

- typedef struct _braid_App_struct * braid_App
- typedef struct
  _braid_Vector_struct * braid_Vector
- typedef braid_Int(* braid_PtFcnPhi)(braid_App app, braid_Vector u, braid_PhiStatus status)
- typedef braid_Int(* braid_PtFcnInit)(braid_App app, braid_Real t, braid_Vector *u_ptr)
- typedef braid_Int(* braid_PtFcnClone)(braid_App app, braid_Vector u, braid_Vector *v_ptr)
- typedef braid_Int(* braid_PtFcnFree)(braid_App app, braid_Vector u)
- typedef braid_Int(* braid_PtFcnSum)(braid_App app, braid_Real alpha, braid_Vector x, braid_Real beta, braid-_Vector y)
- typedef braid_Int(* braid_PtFcnSpatialNorm)(braid_App app, braid_Vector u, braid_Real *norm_ptr)
- typedef braid_Int(* braid_PtFcnAccess)(braid_App app, braid_Vector u, braid_AccessStatus status)
- typedef braid_Int(* braid_PtFcnBufSize)(braid_App app, braid_Int *size_ptr)
- typedef braid_Int(* braid_PtFcnBufPack)(braid_App app, braid_Vector u, void *buffer, braid_Int *size_ptr)
- typedef braid_Int(* braid_PtFcnBufUnpack)(braid_App app, void *buffer, braid_Vector *u_ptr)
- typedef braid_Int(* braid_PtFcnCoarsen)(braid_App app, braid_Vector fu, braid_Vector *cu_ptr, braid_Coarsen-RefStatus status)
- typedef braid_Int(* braid_PtFcnRefine)(braid_App app, braid_Vector cu, braid_Vector *fu_ptr, braid_Coarsen-RefStatus status)

9.1.1 Detailed Description

These are all user-written data structures and routines. There are two data structures (braid_App and braid_Vector) for the user to define. And, there are a variety of function interfaces (defined through function pointer declarations) that the user must implement.

9.1.2 Typedef Documentation

9.1.2.1 typedef struct _braid_App_struct* **braid_App**

This holds a wide variety of information and is `global` in that it is passed to every function. This structure holds everything that the user will need to carry out a simulation. For a simple example, this could just hold the global MPI communicator and a few values describing the temporal domain.

9.1.2.2 typedef braid_Int(* braid_PtFcnAccess)(**braid_App** app,**braid_Vector** u,braid_AccessStatus status)

Gives user access to XBraid and to the current vector *u* at time *t*. Most commonly, this lets the user write the vector to screen, file, etc... The user decides what is appropriate. Note how you are told the time value *t* of the vector *u* and other information in *status*. This lets you tailor the output, e.g., for only certain time values at certain XBraid iterations. Querrying status for such information is done through *braid_AccessStatusGet**(..)* routines.

The frequency of XBraid's calls to *access* is controlled through braid_SetAccessLevel. For instance, if access_level is set to 2, then *access* is called every XBraid iteration and on every XBraid level. In this case, querrying *status* to determine the current XBraid level and iteration will be useful. This scenario allows for even more detailed tracking of the simulation.

Eventually, access will be broadened to allow the user to steer XBraid.

9.1.2.3 typedef braid_Int(∗ braid_PtFcnBufPack)(braid_App app,braid_Vector u,void ∗buffer,braid_Int ∗size_ptr)

This allows XBraid to send messages containing braid_Vectors. This routine packs a vector *u* into a *void ∗ buffer* for MPI.

9.1.2.4 typedef braid_Int(∗ braid_PtFcnBufSize)(braid_App app,braid_Int ∗size_ptr)

This routine tells XBraid message sizes by computing an upper bound in bytes for an arbitrary braid_Vector. This size must be an upper bound for what BufPack and BufUnPack will assume.

9.1.2.5 typedef braid_Int(∗ braid_PtFcnBufUnpack)(braid_App app,void ∗buffer,braid_Vector ∗u_ptr)

This allows XBraid to receive messages containing braid_Vectors. This routine unpacks a *void ∗ buffer* from MPI into a braid_Vector.

9.1.2.6 typedef braid_Int(∗ braid_PtFcnClone)(braid_App app,braid_Vector u,braid_Vector ∗v_ptr)

Clone *u* into *v_ptr*

9.1.2.7 typedef braid_Int(∗ braid_PtFcnCoarsen)(braid_App app,braid_Vector fu,braid_Vector ∗cu_ptr,braid_CoarsenRefStatus status)

Spatial coarsening (optional). Allows the user to coarsen when going from a fine time grid to a coarse time grid. This function is called on every vector at each level, thus you can coarsem the entire space time domain. The action of this function should match the braid_PtFcnRefine function.

The user should query the status structure at run time with *braid_CoarsenRefGet∗∗()* calls in order to determine how to coarsen. For instance, status tells you what the current time value is, and what the time step sizes on the fine and coarse levels are.

9.1.2.8 typedef braid_Int(∗ braid_PtFcnFree)(braid_App app,braid_Vector u)

Free and deallocate *u*

9.1.2.9 typedef braid_Int(∗ braid_PtFcnInit)(braid_App app,braid_Real t,braid_Vector ∗u_ptr)

Initializes a vector *u_ptr* at time *t*

9.1.2.10 typedef braid_Int(∗ braid_PtFcnPhi)(braid_App app,braid_Vector u,braid_PhiStatus status)

Defines the central time stepping function that the user must write. The user must advance the vector *u* from time *tstart* to time *tstop*.

Query the status structure with *braid_PhiStatusGetTstart(status, &tstart)* and *braid_PhiStatusGetTstop(status, &tstop)* to get *tstart* and *tstop*. The status structure also allows for steering. For example, *braid_PhiStatusSetRFactor(...)* allows for setting rfactor, which tells XBraid to refine this time interval.

9.1.2.11 typedef braid_Int(∗ braid_PtFcnRefine)(braid_App app,braid_Vector cu,braid_Vector ∗fu_ptr,braid_CoarsenRefStatus status)

Spatial refinement (optional). Allows the user to refine when going from a coarse time grid to a fine time grid. This function is called on every vector at each level, thus you can refine the entire space time domain. The action of this function should match the braid_PtFcnCoarsen function.

The user should query the status structure at run time with *braid_CoarsenRefGet∗∗()* calls in order to determine how to coarsen. For instance, status tells you what the current time value is, and what the time step sizes on the fine and coarse levels are.

9.1.2.12 typedef braid_Int(∗ braid_PtFcnSpatialNorm)(braid_App app,braid_Vector u,braid_Real ∗norm_ptr)

Carry out a spatial norm by taking the norm of a braid_Vector *norm_ptr* = $||$ *u* $||$ A common choice is the standard Eucliden norm, but many other choices are possible, such as an L2-norm based on a finite element space. See braid_-SetTemporalNorm for information on how the spatial norm is combined over time for a global space-time residual norm. This global norm then controls halting.

9.1.2.13 typedef braid_Int(∗ braid_PtFcnSum)(braid_App app,braid_Real alpha,braid_Vector x,braid_Real beta,braid_Vector y)

AXPY, *alpha x* + *beta y* $->$ *y*

9.1.2.14 typedef struct _braid_Vector_struct∗ braid_Vector

This defines (roughly) a state vector at a certain time value. It could also contain any other information related to this vector which is needed to evolve the vector to the next time value, like mesh information.

9.2 User interface routines

Modules

- General Interface routines
- XBraid status routines

9.2.1 Detailed Description

These are all the user interface routines.

9.3 General Interface routines

Typedefs

- typedef struct _braid_Core_struct ∗ braid_Core

Functions

- braid_Int braid_Init (MPI_Comm comm_world, MPI_Comm comm, braid_Real tstart, braid_Real tstop, braid_Int ntime, braid_App app, braid_PtFcnPhi phi, braid_PtFcnInit init, braid_PtFcnClone clone, braid_PtFcnFree free, braid_PtFcnSum sum, braid_PtFcnSpatialNorm spatialnorm, braid_PtFcnAccess access, braid_PtFcnBufSize bufsize, braid_PtFcnBufPack bufpack, braid_PtFcnBufUnpack bufunpack, braid_Core ∗core_ptr)
- braid_Int braid_Drive (braid_Core core)
- braid_Int braid_Destroy (braid_Core core)
- braid_Int braid_PrintStats (braid_Core core)
- braid_Int braid_SetLoosexTol (braid_Core core, braid_Int level, braid_Real loose_tol)
- braid_Int braid_SetTightxTol (braid_Core core, braid_Int level, braid_Real tight_tol)
- braid_Int braid_SetMaxLevels (braid_Core core, braid_Int max_levels)
- braid_Int braid_SetMinCoarse (braid_Core core, braid_Int min_coarse)
- braid_Int braid_SetAbsTol (braid_Core core, braid_Real atol)
- braid_Int braid_SetRelTol (braid_Core core, braid_Real rtol)
- braid_Int braid_SetNRelax (braid_Core core, braid_Int level, braid_Int nrelax)
- braid_Int braid_SetCFactor (braid_Core core, braid_Int level, braid_Int cfactor)
- braid_Int braid_SetMaxIter (braid_Core core, braid_Int max_iter)
- braid_Int braid_SetFMG (braid_Core core)
- braid_Int braid_SetTemporalNorm (braid_Core core, braid_Int tnorm)
- braid_Int braid_SetNFMGVcyc (braid_Core core, braid_Int nfmg_Vcyc)
- braid_Int braid_SetSpatialCoarsen (braid_Core core, braid_PtFcnCoarsen coarsen)
- braid_Int braid_SetSpatialRefine (braid_Core core, braid_PtFcnRefine refine)
- braid_Int braid_SetPrintLevel (braid_Core core, braid_Int print_level)
- braid_Int braid_SetPrintFile (braid_Core core, const char ∗printfile_name)
- braid_Int braid_SetAccessLevel (braid_Core core, braid_Int access_level)
- braid_Int braid_SplitCommworld (const MPI_Comm ∗comm_world, braid_Int px, MPI_Comm ∗comm_x, MPI_-Comm ∗comm_t)
- braid_Int braid_GetNumIter (braid_Core core, braid_Int ∗niter_ptr)
- braid_Int braid_GetRNorm (braid_Core core, braid_Real ∗rnorm_ptr)
- braid_Int braid_GetNLevels (braid_Core core, braid_Int ∗nlevels_ptr)

9.3.1 Detailed Description

These are general interface routines, e.g., routines to initialize and run a XBraid solver, or to split a communicator into spatial and temporal components.

9.3.2 Typedef Documentation

9.3.2.1 typedef struct _braid_Core_struct∗ **braid_Core**

points to the core structure defined in _braid.h

9.3.3 Function Documentation

9.3.3.1 braid_Int braid_Destroy (braid_Core *core*)

Clean up and destroy core.

Parameters

core	braid_Core (_braid_Core) struct

9.3.3.2 braid_Int braid_Drive (braid_Core *core*)

Carry out a simulation with XBraid. Integrate in time.

Parameters

core	braid_Core (_braid_Core) struct

9.3.3.3 braid_Int braid_GetNLevels (braid_Core *core*, braid_Int ∗ *nlevels_ptr*)

After Drive() finishes, this returns the number of XBraid levels

Parameters

core	braid_Core (_braid_Core) struct
nlevels_ptr	output, holds the number of XBraid levels

9.3.3.4 braid_Int braid_GetNumIter (braid_Core *core*, braid_Int ∗ *niter_ptr*)

After Drive() finishes, this returns the number of iterations taken.

Parameters

core	braid_Core (_braid_Core) struct
niter_ptr	output, holds number of iterations taken

9.3.3.5 braid_Int braid_GetRNorm (braid_Core *core*, braid_Real ∗ *rnorm_ptr*)

After Drive() finishes, this returns the last measured residual norm.

Parameters

core	braid_Core (_braid_Core) struct
rnorm_ptr	output, holds final residual norm

9.3.3.6 braid_Int braid_Init (MPI_Comm *comm_world*, MPI_Comm *comm*, braid_Real *tstart*, braid_Real *tstop*, braid_Int *ntime*, braid_App *app*, braid_PtFcnPhi *phi*, braid_PtFcnInit *init*, braid_PtFcnClone *clone*, braid_PtFcnFree *free*, braid_PtFcnSum *sum*, braid_PtFcnSpatialNorm *spatialnorm*, braid_PtFcnAccess *access*, braid_PtFcnBufSize *bufsize*, braid_PtFcnBufPack *bufpack*, braid_PtFcnBufUnpack *bufunpack*, braid_Core ∗ *core_ptr*)

Create a core object with the required initial data.

This core is used by XBraid for internal data structures. The output is *core_ptr* which points to the newly created braid_Core structure.

Parameters

comm_world	Global communicator for space and time

comm	Communicator for temporal dimension
tstart	start time
tstop	End time
ntime	Initial number of temporal grid values
app	User-defined _braid_App structure
phi	User time stepping routine to advance a braid_Vector forward one step
init	Initialize a braid_Vector on the finest temporal grid
clone	Clone a braid_Vector
free	Free a braid_Vector
sum	Compute vector sum of two braid_Vectors
spatialnorm	Compute norm of a braid_Vector, this is a norm only over space
access	Allows access to XBraid and current braid_Vector
bufsize	Computes size for MPI buffer for one braid_Vector
bufpack	Packs MPI buffer to contain one braid_Vector
bufunpack	Unpacks MPI buffer into a braid_Vector
core_ptr	Pointer to braid_Core (_braid_Core) struct

9.3.3.7 braid_Int braid_PrintStats (braid_Core *core*)

Print statistics after a XBraid run.

Parameters

core	braid_Core (_braid_Core) struct

9.3.3.8 braid_Int braid_SetAbsTol (braid_Core *core*, braid_Real *atol*)

Set absolute stopping tolerance.

Recommended option over relative tolerance

Parameters

core	braid_Core (_braid_Core) struct
atol	absolute stopping tolerance

9.3.3.9 braid_Int braid_SetAccessLevel (braid_Core *core*, braid_Int *access_level*)

Set access level for XBraid. This controls how often the user's access routine is called.

- Level 0: Never call the user's access routine
- Level 1: Only call the user's access routine after XBraid is finished
- Level 2: Call the user's access routine every iteration and on every level. This is during _braid_FRestrict, during the down-cycle part of a XBraid iteration.

Default is level 1.

Parameters

core	braid_Core (_braid_Core) struct

access_level	desired access_level

9.3.3.10 braid_Int braid_SetCFactor (braid_Core *core,* braid_Int *level,* braid_Int *cfactor*)

Set the coarsening factor *cfactor* on grid *level* (level 0 is the finest grid). The default factor is 2 on all levels. To change the default factor, use *level = -1*.

Parameters

core	braid_Core (_braid_Core) struct
level	*level* to set coarsening factor on
cfactor	desired coarsening factor

9.3.3.11 braid_Int braid_SetFMG (braid_Core *core*)

Once called, XBraid will use FMG (i.e., F-cycles.

Parameters

core	braid_Core (_braid_Core) struct

9.3.3.12 braid_Int braid_SetLoosexTol (braid_Core *core,* braid_Int *level,* braid_Real *loose_tol*)

Set loose stopping tolerance *loose_tol* for spatial solves on grid *level* (level 0 is the finest grid).

Parameters

core	braid_Core (_braid_Core) struct
level	level to set *loose_tol*
loose_tol	tolerance to set

9.3.3.13 braid_Int braid_SetMaxIter (braid_Core *core,* braid_Int *max_iter*)

Set max number of multigrid iterations.

Parameters

core	braid_Core (_braid_Core) struct
max_iter	maximum iterations to allow

9.3.3.14 braid_Int braid_SetMaxLevels (braid_Core *core,* braid_Int *max_levels*)

Set max number of multigrid levels.

Parameters

core	braid_Core (_braid_Core) struct
max_levels	maximum levels to allow

9.3.3.15 braid_Int braid_SetMinCoarse (braid_Core *core,* braid_Int *min_coarse*)

Set minimum allowed coarse grid size. XBraid stops coarsening whenever creating the next coarser grid will result in a grid smaller than min_coarse. The maximum possible coarse grid size will be min_coarse$*$coarsening_factor.

Parameters

core	braid_Core (_braid_Core) struct
min_coarse	minimum coarse grid size

9.3.3.16 braid_Int braid_SetNFMGVcyc (braid_Core *core*, braid_Int *nfmg_Vcyc*)

Set number of V-cycles to use at each FMG level (standard is 1)

Parameters

core	braid_Core (_braid_Core) struct
nfmg_Vcyc	number of V-cycles to do each FMG level

9.3.3.17 braid_Int braid_SetNRelax (braid_Core *core*, braid_Int *level*, braid_Int *nrelax*)

Set the number of relaxation sweeps *nrelax* on grid *level* (level 0 is the finest grid). The default is 1 on all levels. To change the default factor, use *level = -1*. One sweep is a CF relaxation sweep.

Parameters

core	braid_Core (_braid_Core) struct
level	*level* to set *nrelax* on
nrelax	number of relaxations to do on *level*

9.3.3.18 braid_Int braid_SetPrintFile (braid_Core *core*, const char * *printfile_name*)

Set output file for runtime print messages. Level of printing is controlled by braid_SetPrintLevel. Default is stdout.

Parameters

core	braid_Core (_braid_Core) struct
printfile_name	output file for XBraid runtime output

9.3.3.19 braid_Int braid_SetPrintLevel (braid_Core *core*, braid_Int *print_level*)

Set print level for XBraid. This controls how much information is printed to the XBraid print file (braid_SetPrintFile).

- Level 0: no output
- Level 1: print typical information like a residual history, number of levels in the XBraid hierarchy, and so on.
- Level 2: level 1 output, plus debug level output.

Default is level 1.

Parameters

core	braid_Core (_braid_Core) struct
print_level	desired print level

9.3.3.20 braid_Int braid_SetRelTol (braid_Core *core*, braid_Real *rtol*)

Set relative stopping tolerance, relative to the initial residual. Be careful. If your initial guess is all zero, then the initial residual may only be nonzero over one or two time values, and this will skew the relative tolerance. Absolute tolerances are recommended.

Parameters

core	braid_Core (_braid_Core) struct
rtol	relative stopping tolerance

9.3.3.21 braid_Int braid_SetSpatialCoarsen (braid_Core *core*, braid_PtFcnCoarsen *coarsen*)

Set spatial coarsening routine with user-defined routine. Default is no spatial refinment or coarsening.

Parameters

core	braid_Core (_braid_Core) struct
coarsen	function pointer to spatial coarsening routine

9.3.3.22 braid_Int braid_SetSpatialRefine (braid_Core *core*, braid_PtFcnRefine *refine*)

Set spatial refinement routine with user-defined routine. Default is no spatial refinment or coarsening.

Parameters

core	braid_Core (_braid_Core) struct
refine	function pointer to spatial refinement routine

9.3.3.23 braid_Int braid_SetTemporalNorm (braid_Core *core*, braid_Int *tnorm*)

Sets XBraid temporal norm.

This option determines how to obtain a global space-time residual norm. That is, this decides how to combine the spatial norms returned by braid_PtFcnSpatialNorm at each time step to obtain a global norm over space and time. It is this global norm that then controls halting.

There are three options for setting *tnorm*. See section Halting tolerance for a more detailed discussion (in Introduction.-md).

- *tnorm=1*: One-norm summation of spatial norms
- *tnorm=2*: Two-norm summation of spatial norms
- *tnorm=3*: Infinity-norm combination of spatial norms

The default choice is *tnorm=2*

Parameters

core	braid_Core (_braid_Core) struct
tnorm	choice of temporal norm

9.3.3.24 braid_Int braid_SetTightxTol (braid_Core *core*, braid_Int *level*, braid_Real *tight_tol*)

Set tight stopping tolerance *tight_tol* for spatial solves on grid *level* (level 0 is the finest grid).

Parameters

core	braid_Core (_braid_Core) struct

level	level to set *tight_tol*
tight_tol	tolerance to set

9.3.3.25 braid_Int braid_SplitCommworld (const MPI_Comm * *comm_world,* braid_Int *px,* MPI_Comm * *comm_x,* MPI_Comm * *comm_t*)

Split MPI commworld into *comm_x* and *comm_t*, the spatial and temporal communicators. The total number of processors will equal Px*Pt, there Px is the number of procs in space, and Pt is the number of procs in time.

Parameters

comm_world	Global communicator to split
px	Number of processors parallelizing space for a single time step
comm_x	Spatial communicator (written as output)
comm_t	Temporal communicator (written as output)

9.4 XBraid status routines

Functions

- braid_Int braid_AccessStatusGetT (braid_AccessStatus status, braid_Real ∗t_ptr)
- braid_Int braid_AccessStatusGetResidual (braid_AccessStatus status, braid_Real ∗rnorm_ptr)
- braid_Int braid_AccessStatusGetIter (braid_AccessStatus status, braid_Int ∗iter_ptr)
- braid_Int braid_AccessStatusGetLevel (braid_AccessStatus status, braid_Int ∗level_ptr)
- braid_Int braid_AccessStatusGetDone (braid_AccessStatus status, braid_Int ∗done_ptr)
- braid_Int braid_AccessStatusGetWrapperTest (braid_AccessStatus status, braid_Int ∗wtest_ptr)
- braid_Int braid_AccessStatusGetTILD (braid_AccessStatus status, braid_Real ∗t_ptr, braid_Int ∗iter_ptr, braid_Int ∗level_ptr, braid_Int ∗done_ptr)
- braid_Int braid_CoarsenRefStatusGetTstart (braid_CoarsenRefStatus status, braid_Real ∗tstart_ptr)
- braid_Int braid_CoarsenRefStatusGetFTstop (braid_CoarsenRefStatus status, braid_Real ∗f_tstop_ptr)
- braid_Int braid_CoarsenRefStatusGetFTprior (braid_CoarsenRefStatus status, braid_Real ∗f_tprior_ptr)
- braid_Int braid_CoarsenRefStatusGetCTstop (braid_CoarsenRefStatus status, braid_Real ∗c_tstop_ptr)
- braid_Int braid_CoarsenRefStatusGetCTprior (braid_CoarsenRefStatus status, braid_Real ∗c_tprior_ptr)
- braid_Int braid_CoarsenRefStatusGetTpriorTstop (braid_CoarsenRefStatus status, braid_Real ∗tstart_ptr, braid_Real ∗f_tprior_ptr, braid_Real ∗f_tstop_ptr, braid_Real ∗c_tprior_ptr, braid_Real ∗c_tstop_ptr)
- braid_Int braid_CoarsenRefStatusGetLevel (braid_CoarsenRefStatus status, braid_Int ∗level_ptr)
- braid_Int braid_PhiStatusGetTstart (braid_PhiStatus status, braid_Real ∗tstart_ptr)
- braid_Int braid_PhiStatusGetTstop (braid_PhiStatus status, braid_Real ∗tstop_ptr)
- braid_Int braid_PhiStatusGetAccuracy (braid_PhiStatus status, braid_Real ∗accuracy_ptr)
- braid_Int braid_PhiStatusGetLevel (braid_PhiStatus status, braid_Int ∗level_ptr)
- braid_Int braid_PhiStatusSetRFactor (braid_PhiStatus status, braid_Real rfactor)
- braid_Int braid_PhiStatusGetTstartTstop (braid_PhiStatus status, braid_Real ∗tstart_ptr, braid_Real ∗tstop_ptr)

9.4.1 Detailed Description

XBraid status structures are what tell the user the status of the simulation when their routines (phi, coarsen/refine, access) are called.

9.4.2 Function Documentation

9.4.2.1 braid_Int braid_AccessStatusGetDone (braid_AccessStatus *status,* braid_Int ∗ *done_ptr*)

Return whether XBraid is done for the current simulation.

done_ptr = 1 indicates that XBraid has finished iterating, (either maxiter has been reached, or the tolerance has been met).

Parameters

status	structure containing current simulation info
done_ptr	output, =1 if XBraid has finished, else =0

9.4.2.2 braid_Int braid_AccessStatusGetIter (braid_AccessStatus *status,* braid_Int ∗ *iter_ptr*)

Return the current iteration from the AccessStatus structure.

Parameters

status	structure containing current simulation info
iter_ptr	output, current XBraid iteration number

9.4.2.3 braid_Int braid_AccessStatusGetLevel (braid_AccessStatus *status,* braid_Int ∗ *level_ptr*)

Return the current XBraid level from the AccessStatus structure.

Parameters

status	structure containing current simulation info
level_ptr	output, current level in XBraid

9.4.2.4 braid_Int braid_AccessStatusGetResidual (braid_AccessStatus *status,* braid_Real ∗ *rnorm_ptr*)

Return the current residual norm from the AccessStatus structure.

Parameters

status	structure containing current simulation info
rnorm_ptr	output, current residual norm

9.4.2.5 braid_Int braid_AccessStatusGetT (braid_AccessStatus *status,* braid_Real ∗ *t_ptr*)

Return the current time from the AccessStatus structure.

Parameters

status	structure containing current simulation info
t_ptr	output, current time

9.4.2.6 braid_Int braid_AccessStatusGetTILD (braid_AccessStatus *status,* braid_Real ∗ *t_ptr,* braid_Int ∗ *iter_ptr,* braid_Int ∗ *level_ptr,* braid_Int ∗ *done_ptr*)

Return XBraid status for the current simulation. Four values are returned.

TILD : time, iteration, level, done

These values are also available through individual Get routines. These individual routines are the location of detailed documentation on each parameter, e.g., see *braid_AccessStatusGetDone* for more information on the *done* value.

Parameters

status	structure containing current simulation info
t_ptr	output, current time
iter_ptr	output, current iteration in XBraid
level_ptr	output, current level in XBraid
done_ptr	output, boolean describing whether XBraid has finished

9.4.2.7 braid_Int braid_AccessStatusGetWrapperTest (braid_AccessStatus *status,* braid_Int ∗ *wtest_ptr*)

Return whether this is a wrapper test or an XBraid run

Parameters

status	structure containing current simulation info
wtest_ptr	output, =1 if this is a wrapper test, =0 if XBraid run

9.4.2.8 braid_Int braid_CoarsenRefStatusGetCTprior (braid_CoarsenRefStatus *status*, braid_Real ∗ *c_tprior_ptr*)

Return the **coarse grid** time value to the left of the current time value from the CoarsenRefStatus structure.

Parameters

status	structure containing current simulation info
c_tprior_ptr	output, time value to the left of current time value on coarse grid

9.4.2.9 braid_Int braid_CoarsenRefStatusGetCTstop (braid_CoarsenRefStatus *status*, braid_Real ∗ *c_tstop_ptr*)

Return the **coarse grid** time value to the right of the current time value from the CoarsenRefStatus structure.

Parameters

status	structure containing current simulation info
c_tstop_ptr	output, time value to the right of current time value on coarse grid

9.4.2.10 braid_Int braid_CoarsenRefStatusGetFTprior (braid_CoarsenRefStatus *status*, braid_Real ∗ *f_tprior_ptr*)

Return the **fine grid** time value to the left of the current time value from the CoarsenRefStatus structure.

Parameters

status	structure containing current simulation info
f_tprior_ptr	output, time value to the left of current time value on fine grid

9.4.2.11 braid_Int braid_CoarsenRefStatusGetFTstop (braid_CoarsenRefStatus *status*, braid_Real ∗ *f_tstop_ptr*)

Return the **fine grid** time value to the right of the current time value from the CoarsenRefStatus structure.

Parameters

status	structure containing current simulation info
f_tstop_ptr	output, time value to the right of current time value on fine grid

9.4.2.12 braid_Int braid_CoarsenRefStatusGetLevel (braid_CoarsenRefStatus *status*, braid_Int ∗ *level_ptr*)

Return the current XBraid level from the CoarsenRefStatus structure.

Parameters

status	structure containing current simulation info
level_ptr	output, current fine level in XBraid

9.4.2.13 braid_Int braid_CoarsenRefStatusGetTpriorTstop (braid_CoarsenRefStatus *status*, braid_Real ∗ *tstart_ptr*, braid_Real ∗ *f_tprior_ptr*, braid_Real ∗ *f_tstop_ptr*, braid_Real ∗ *c_tprior_ptr*, braid_Real ∗ *c_tstop_ptr*)

Return XBraid status for the current simulation. Five values are returned, tstart, f_tprior, f_tstop, c_tprior, c_tstop.

These values are also available through individual Get routines. These individual routines are the location of detailed documentation on each parameter, e.g., see *braid_CoarsenRefStatusGetCTprior* for more information on the *c_tprior* value.

Parameters

status	structure containing current simulation info
tstart_ptr	output, time value current vector
f_tprior_ptr	output, time value to the left of tstart on fine grid
f_tstop_ptr	output, time value to the right of tstart on fine grid
c_tprior_ptr	output, time value to the left of tstart on coarse grid
c_tstop_ptr	output, time value to the right of tstart on coarse grid

9.4.2.14 braid_Int braid_CoarsenRefStatusGetTstart (braid_CoarsenRefStatus *status,* braid_Real ∗ *tstart_ptr*)

Return the current time value from the CoarsenRefStatus structure.

Parameters

status	structure containing current simulation info
tstart_ptr	output, current time

9.4.2.15 braid_Int braid_PhiStatusGetAccuracy (braid_PhiStatus *status,* braid_Real ∗ *accuracy_ptr*)

Return the current accuracy value, usually between 0 and 1.0, which can allow for tuning of implicit solve accuracy

Parameters

status	structure containing current simulation info
accuracy_ptr	output, current accuracy value

9.4.2.16 braid_Int braid_PhiStatusGetLevel (braid_PhiStatus *status,* braid_Int ∗ *level_ptr*)

Return the current XBraid level from the PhiStatus structure.

Parameters

status	structure containing current simulation info
level_ptr	output, current level in XBraid

9.4.2.17 braid_Int braid_PhiStatusGetTstart (braid_PhiStatus *status,* braid_Real ∗ *tstart_ptr*)

Return the current time value from the PhiStatus structure.

Parameters

status	structure containing current simulation info
tstart_ptr	output, current time

9.4.2.18 braid_Int braid_PhiStatusGetTstartTstop (braid_PhiStatus *status,* braid_Real ∗ *tstart_ptr,* braid_Real ∗ *tstop_ptr*)

Return XBraid status for the current simulation. Two values are returned, tstart and tstop.

These values are also available through individual Get routines. These individual routines are the location of detailed documentation on each parameter, e.g., see *braid_PhiStatusGetTstart* for more information on the *tstart* value.

Parameters

status	structure containing current simulation info
tstart_ptr	output, current time
tstop_ptr	output, next time value to evolve towards

9.4.2.19 braid_Int braid_PhiStatusGetTstop (braid_PhiStatus *status*, braid_Real ∗ *tstop_ptr*)

Return the time value to the right of the current time value from the PhiStatus structure.

Parameters

status	structure containing current simulation info
tstop_ptr	output, next time value to evolve towards

9.4.2.20 braid_Int braid_PhiStatusSetRFactor (braid_PhiStatus *status*, braid_Real *rfactor*)

Set the rfactor, a desired refinement factor for this interval. rfactor=1 indicates no refinement, otherwise, this inteval is subdivided rfactor times.

Parameters

status	structure containing current simulation info
rfactor	user-determined desired rfactor

9.5 XBraid test routines

Functions

- braid_Int braid_TestInitAccess (braid_App app, MPI_Comm comm_x, FILE ∗fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnAccess access, braid_PtFcnFree free)
- braid_Int braid_TestClone (braid_App app, MPI_Comm comm_x, FILE ∗fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnAccess access, braid_PtFcnFree free, braid_PtFcnClone clone)
- braid_Int braid_TestSum (braid_App app, MPI_Comm comm_x, FILE ∗fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnAccess access, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum)
- braid_Int braid_TestSpatialNorm (braid_App app, MPI_Comm comm_x, FILE ∗fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum, braid_PtFcnSpatialNorm spatialnorm)
- braid_Int braid_TestBuf (braid_App app, MPI_Comm comm_x, FILE ∗fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnFree free, braid_PtFcnSum sum, braid_PtFcnSpatialNorm spatialnorm, braid_PtFcnBufSize bufsize, braid_PtFcnBufPack bufpack, braid_PtFcnBufUnpack bufunpack)
- braid_Int braid_TestCoarsenRefine (braid_App app, MPI_Comm comm_x, FILE ∗fp, braid_Real t, braid_Real fdt, braid_Real cdt, braid_PtFcnInit init, braid_PtFcnAccess access, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum, braid_PtFcnSpatialNorm spatialnorm, braid_PtFcnCoarsen coarsen, braid_PtFcnRefine refine)
- braid_Int braid_TestAll (braid_App app, MPI_Comm comm_x, FILE ∗fp, braid_Real t, braid_Real fdt, braid_Real cdt, braid_PtFcnInit init, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum, braid_PtFcnSpatialNorm spatialnorm, braid_PtFcnBufSize bufsize, braid_PtFcnBufPack bufpack, braid_PtFcnBufUnpack bufunpack, braid_PtFcnCoarsen coarsen, braid_PtFcnRefine refine)

9.5.1 Detailed Description

These are sanity check routines to help a user test their XBraid code.

9.5.2 Function Documentation

9.5.2.1 braid_Int braid_TestAll (**braid_App** *app*, MPI_Comm *comm_x*, FILE ∗ *fp*, braid_Real *t*, braid_Real *fdt*, braid_Real *cdt*, **braid_PtFcnInit** *init*, **braid_PtFcnFree** *free*, **braid_PtFcnClone** *clone*, **braid_PtFcnSum** *sum*, **braid_PtFcnSpatialNorm** *spatialnorm*, **braid_PtFcnBufSize** *bufsize*, **braid_PtFcnBufPack** *bufpack*, **braid_PtFcnBufUnpack** *bufunpack*, **braid_PtFcnCoarsen** *coarsen*, **braid_PtFcnRefine** *refine*)

Runs all of the individual braid_Test∗ routines

- Returns 0 if the tests fail
- Returns 1 if the tests pass
- Check the log messages to see details of which tests failed.

Parameters

app	User defined App structure
comm_x	Spatial communicator
fp	File pointer (could be stdout or stderr) for log messages

t	Time value to initialize test vectors with
fdt	Fine time step value that you spatially coarsen from
cdt	Coarse time step value that you coarsen to
init	Initialize a braid_Vector on finest temporal grid
free	Free a braid_Vector
clone	Clone a braid_Vector
sum	Compute vector sum of two braid_Vectors
spatialnorm	Compute norm of a braid_Vector, this is a norm only over space
bufsize	Computes size in bytes for one braid_Vector MPI buffer
bufpack	Packs MPI buffer to contain one braid_Vector
bufunpack	Unpacks MPI buffer into a braid_Vector
coarsen	Spatially coarsen a vector. If NULL, test is skipped.
refine	Spatially refine a vector. If NULL, test is skipped.

9.5.2.2 braid_Int braid_TestBuf (braid_App *app*, MPI_Comm *comm_x*, FILE ∗ *fp*, braid_Real *t*, braid_PtFcnInit *init*, braid_PtFcnFree *free*, braid_PtFcnSum *sum*, braid_PtFcnSpatialNorm *spatialnorm*, braid_PtFcnBufSize *bufsize*, braid_PtFcnBufPack *bufpack*, braid_PtFcnBufUnpack *bufunpack*)

Test the BufPack, BufUnpack and BufSize functions.

A vector is initialized at time *t*, packed into a buffer, then unpacked from a buffer. The unpacked result must equal the original vector.

- Returns 0 if the tests fail
- Returns 1 if the tests pass
- Check the log messages to see details of which tests failed.

Parameters

app	User defined App structure
comm_x	Spatial communicator
fp	File pointer (could be stdout or stderr) for log messages
t	Time value to test Buffer routines (used to initialize the vectors)
init	Initialize a braid_Vector on finest temporal grid
free	Free a braid_Vector
sum	Compute vector sum of two braid_Vectors
spatialnorm	Compute norm of a braid_Vector, this is a norm only over space
bufsize	Computes size in bytes for one braid_Vector MPI buffer
bufpack	Packs MPI buffer to contain one braid_Vector
bufunpack	Unpacks MPI buffer containing one braid_Vector

9.5.2.3 braid_Int braid_TestClone (braid_App *app*, MPI_Comm *comm_x*, FILE ∗ *fp*, braid_Real *t*, braid_PtFcnInit *init*, braid_PtFcnAccess *access*, braid_PtFcnFree *free*, braid_PtFcnClone *clone*)

Test the clone function.

A vector is initialized at time *t*, cloned, and both vectors are written. Then both vectors are free-d. The user is to check (via the access function) to see if it is identical.

Parameters

app	User defined App structure
comm_x	Spatial communicator
fp	File pointer (could be stdout or stderr) for log messages
t	Time value to test clone with (used to initialize the vectors)
init	Initialize a braid_Vector on finest temporal grid
access	Allows access to XBraid and current braid_Vector (can be NULL for no writing)
free	Free a braid_Vector
clone	Clone a braid_Vector

9.5.2.4 braid_Int braid_TestCoarsenRefine (braid_App *app*, MPI_Comm *comm_x*, FILE ∗ *fp*, braid_Real *t*, braid_Real *fdt*, braid_Real *cdt*, braid_PtFcnInit *init*, braid_PtFcnAccess *access*, braid_PtFcnFree *free*, braid_PtFcnClone *clone*, braid_PtFcnSum *sum*, braid_PtFcnSpatialNorm *spatialnorm*, braid_PtFcnCoarsen *coarsen*, braid_PtFcnRefine *refine*)

Test the Coarsen and Refine functions.

A vector is initialized at time *t*, and various sanity checks on the spatial coarsening and refinement routines are run.

- Returns 0 if the tests fail
- Returns 1 if the tests pass
- Check the log messages to see details of which tests failed.

Parameters

app	User defined App structure
comm_x	Spatial communicator
fp	File pointer (could be stdout or stderr) for log messages
t	Time value to initialize test vectors
fdt	Fine time step value that you spatially coarsen from
cdt	Coarse time step value that you coarsen to
init	Initialize a braid_Vector on finest temporal grid
access	Allows access to XBraid and current braid_Vector (can be NULL for no writing)
free	Free a braid_Vector
clone	Clone a braid_Vector
sum	Compute vector sum of two braid_Vectors
spatialnorm	Compute norm of a braid_Vector, this is a norm only over space
coarsen	Spatially coarsen a vector
refine	Spatially refine a vector

9.5.2.5 braid_Int braid_TestInitAccess (braid_App *app*, MPI_Comm *comm_x*, FILE ∗ *fp*, braid_Real *t*, braid_PtFcnInit *init*, braid_PtFcnAccess *access*, braid_PtFcnFree *free*)

Test the init, access and free functions.

A vector is initialized at time *t*, written, and then free-d

Parameters

app	User defined App structure
comm_x	Spatial communicator
fp	File pointer (could be stdout or stderr) for log messages
t	Time value to test init with (used to initialize the vectors)
init	Initialize a braid_Vector on finest temporal grid
access	Allows access to XBraid and current braid_Vector (can be NULL for no writing)
free	Free a braid_Vector

9.5.2.6 braid_Int braid_TestSpatialNorm (**braid_App** *app*, MPI_Comm *comm_x*, FILE ∗ *fp*, braid_Real *t*, **braid_PtFcnInit** *init*, **braid_PtFcnFree** *free*, **braid_PtFcnClone** *clone*, **braid_PtFcnSum** *sum*, **braid_PtFcnSpatialNorm** *spatialnorm*)

Test the spatialnorm function.

A vector is initialized at time *t* and then cloned. Various norm evaluations like || 3 v || / || v || with known output are then done.

- Returns 0 if the tests fail
- Returns 1 if the tests pass
- Check the log messages to see details of which tests failed.

Parameters

app	User defined App structure
comm_x	Spatial communicator
fp	File pointer (could be stdout or stderr) for log messages
t	Time value to test SpatialNorm with (used to initialize the vectors)
init	Initialize a braid_Vector on finest temporal grid
free	Free a braid_Vector
clone	Clone a braid_Vector
sum	Compute vector sum of two braid_Vectors
spatialnorm	Compute norm of a braid_Vector, this is a norm only over space

9.5.2.7 braid_Int braid_TestSum (**braid_App** *app*, MPI_Comm *comm_x*, FILE ∗ *fp*, braid_Real *t*, **braid_PtFcnInit** *init*, **braid_PtFcnAccess** *access*, **braid_PtFcnFree** *free*, **braid_PtFcnClone** *clone*, **braid_PtFcnSum** *sum*)

Test the sum function.

A vector is initialized at time *t*, cloned, and then these two vectors are summed a few times, with the results written. The vectors are then free-d. The user is to check (via the access function) that the output matches the sum of the two original vectors.

Parameters

app	User defined App structure
comm_x	Spatial communicator
fp	File pointer (could be stdout or stderr) for log messages
t	Time value to test Sum with (used to initialize the vectors)
init	Initialize a braid_Vector on finest temporal grid

access	Allows access to XBraid and current braid_Vector (can be NULL for no writing)
free	Free a braid_Vector
clone	Clone a braid_Vector
sum	Compute vector sum of two braid_Vectors

10 File Documentation

10.1 braid.h File Reference

Typedefs

- typedef struct _braid_App_struct ∗ braid_App
- typedef struct
  _braid_Vector_struct ∗ braid_Vector
- typedef braid_Int(∗ braid_PtFcnPhi)(braid_App app, braid_Vector u, braid_PhiStatus status)
- typedef braid_Int(∗ braid_PtFcnInit)(braid_App app, braid_Real t, braid_Vector ∗u_ptr)
- typedef braid_Int(∗ braid_PtFcnClone)(braid_App app, braid_Vector u, braid_Vector ∗v_ptr)
- typedef braid_Int(∗ braid_PtFcnFree)(braid_App app, braid_Vector u)
- typedef braid_Int(∗ braid_PtFcnSum)(braid_App app, braid_Real alpha, braid_Vector x, braid_Real beta, braid_Vector y)
- typedef braid_Int(∗ braid_PtFcnSpatialNorm)(braid_App app, braid_Vector u, braid_Real ∗norm_ptr)
- typedef braid_Int(∗ braid_PtFcnAccess)(braid_App app, braid_Vector u, braid_AccessStatus status)
- typedef braid_Int(∗ braid_PtFcnBufSize)(braid_App app, braid_Int ∗size_ptr)
- typedef braid_Int(∗ braid_PtFcnBufPack)(braid_App app, braid_Vector u, void ∗buffer, braid_Int ∗size_ptr)
- typedef braid_Int(∗ braid_PtFcnBufUnpack)(braid_App app, void ∗buffer, braid_Vector ∗u_ptr)
- typedef braid_Int(∗ braid_PtFcnCoarsen)(braid_App app, braid_Vector fu, braid_Vector ∗cu_ptr, braid_CoarsenRefStatus status)
- typedef braid_Int(∗ braid_PtFcnRefine)(braid_App app, braid_Vector cu, braid_Vector ∗fu_ptr, braid_CoarsenRefStatus status)
- typedef struct _braid_Core_struct ∗ braid_Core

Functions

- braid_Int braid_Init (MPI_Comm comm_world, MPI_Comm comm, braid_Real tstart, braid_Real tstop, braid_Int ntime, braid_App app, braid_PtFcnPhi phi, braid_PtFcnInit init, braid_PtFcnClone clone, braid_PtFcnFree free, braid_PtFcnSum sum, braid_PtFcnSpatialNorm spatialnorm, braid_PtFcnAccess access, braid_PtFcnBufSize bufsize, braid_PtFcnBufPack bufpack, braid_PtFcnBufUnpack bufunpack, braid_Core ∗core_ptr)
- braid_Int braid_Drive (braid_Core core)
- braid_Int braid_Destroy (braid_Core core)
- braid_Int braid_PrintStats (braid_Core core)
- braid_Int braid_SetLoosexTol (braid_Core core, braid_Int level, braid_Real loose_tol)
- braid_Int braid_SetTightxTol (braid_Core core, braid_Int level, braid_Real tight_tol)
- braid_Int braid_SetMaxLevels (braid_Core core, braid_Int max_levels)
- braid_Int braid_SetMinCoarse (braid_Core core, braid_Int min_coarse)
- braid_Int braid_SetAbsTol (braid_Core core, braid_Real atol)
- braid_Int braid_SetRelTol (braid_Core core, braid_Real rtol)
- braid_Int braid_SetNRelax (braid_Core core, braid_Int level, braid_Int nrelax)
- braid_Int braid_SetCFactor (braid_Core core, braid_Int level, braid_Int cfactor)
- braid_Int braid_SetMaxIter (braid_Core core, braid_Int max_iter)
- braid_Int braid_SetFMG (braid_Core core)
- braid_Int braid_SetTemporalNorm (braid_Core core, braid_Int tnorm)
- braid_Int braid_SetNFMGVcyc (braid_Core core, braid_Int nfmg_Vcyc)
- braid_Int braid_SetSpatialCoarsen (braid_Core core, braid_PtFcnCoarsen coarsen)
- braid_Int braid_SetSpatialRefine (braid_Core core, braid_PtFcnRefine refine)
- braid_Int braid_SetPrintLevel (braid_Core core, braid_Int print_level)

- braid_Int braid_SetPrintFile (braid_Core core, const char ∗printfile_name)
- braid_Int braid_SetAccessLevel (braid_Core core, braid_Int access_level)
- braid_Int braid_SplitCommworld (const MPI_Comm ∗comm_world, braid_Int px, MPI_Comm ∗comm_x, MPI_-Comm ∗comm_t)
- braid_Int braid_GetNumIter (braid_Core core, braid_Int ∗niter_ptr)
- braid_Int braid_GetRNorm (braid_Core core, braid_Real ∗rnorm_ptr)
- braid_Int braid_GetNLevels (braid_Core core, braid_Int ∗nlevels_ptr)

10.1.1 Detailed Description

Define headers for user interface routines. This file contains routines used to allow the user to initialize, run and get and set a XBraid solver.

10.2 braid_status.h File Reference

Functions

- braid_Int braid_AccessStatusGetT (braid_AccessStatus status, braid_Real ∗t_ptr)
- braid_Int braid_AccessStatusGetResidual (braid_AccessStatus status, braid_Real ∗rnorm_ptr)
- braid_Int braid_AccessStatusGetIter (braid_AccessStatus status, braid_Int ∗iter_ptr)
- braid_Int braid_AccessStatusGetLevel (braid_AccessStatus status, braid_Int ∗level_ptr)
- braid_Int braid_AccessStatusGetDone (braid_AccessStatus status, braid_Int ∗done_ptr)
- braid_Int braid_AccessStatusGetWrapperTest (braid_AccessStatus status, braid_Int ∗wtest_ptr)
- braid_Int braid_AccessStatusGetTILD (braid_AccessStatus status, braid_Real ∗t_ptr, braid_Int ∗iter_ptr, braid_-Int ∗level_ptr, braid_Int ∗done_ptr)
- braid_Int braid_CoarsenRefStatusGetTstart (braid_CoarsenRefStatus status, braid_Real ∗tstart_ptr)
- braid_Int braid_CoarsenRefStatusGetFTstop (braid_CoarsenRefStatus status, braid_Real ∗f_tstop_ptr)
- braid_Int braid_CoarsenRefStatusGetFTprior (braid_CoarsenRefStatus status, braid_Real ∗f_tprior_ptr)
- braid_Int braid_CoarsenRefStatusGetCTstop (braid_CoarsenRefStatus status, braid_Real ∗c_tstop_ptr)
- braid_Int braid_CoarsenRefStatusGetCTprior (braid_CoarsenRefStatus status, braid_Real ∗c_tprior_ptr)
- braid_Int braid_CoarsenRefStatusGetTpriorTstop (braid_CoarsenRefStatus status, braid_Real ∗tstart_ptr, braid-_Real ∗f_tprior_ptr, braid_Real ∗f_tstop_ptr, braid_Real ∗c_tprior_ptr, braid_Real ∗c_tstop_ptr)
- braid_Int braid_CoarsenRefStatusGetLevel (braid_CoarsenRefStatus status, braid_Int ∗level_ptr)
- braid_Int braid_PhiStatusGetTstart (braid_PhiStatus status, braid_Real ∗tstart_ptr)
- braid_Int braid_PhiStatusGetTstop (braid_PhiStatus status, braid_Real ∗tstop_ptr)
- braid_Int braid_PhiStatusGetAccuracy (braid_PhiStatus status, braid_Real ∗accuracy_ptr)
- braid_Int braid_PhiStatusGetLevel (braid_PhiStatus status, braid_Int ∗level_ptr)
- braid_Int braid_PhiStatusSetRFactor (braid_PhiStatus status, braid_Real rfactor)
- braid_Int braid_PhiStatusGetTstartTstop (braid_PhiStatus status, braid_Real ∗tstart_ptr, braid_Real ∗tstop_ptr)

10.2.1 Detailed Description

Define headers for XBraid status structures, status get/set routines and status create/destroy routines.

10.3 braid_test.h File Reference

Functions

- braid_Int braid_TestInitAccess (braid_App app, MPI_Comm comm_x, FILE ∗fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnAccess access, braid_PtFcnFree free)
- braid_Int braid_TestClone (braid_App app, MPI_Comm comm_x, FILE ∗fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnAccess access, braid_PtFcnFree free, braid_PtFcnClone clone)
- braid_Int braid_TestSum (braid_App app, MPI_Comm comm_x, FILE ∗fp, braid_Real t, braid_PtFcnInit init, braid-_PtFcnAccess access, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum)
- braid_Int braid_TestSpatialNorm (braid_App app, MPI_Comm comm_x, FILE ∗fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum, braid_PtFcnSpatialNorm spatialnorm)
- braid_Int braid_TestBuf (braid_App app, MPI_Comm comm_x, FILE ∗fp, braid_Real t, braid_PtFcnInit init, braid_-PtFcnFree free, braid_PtFcnSum sum, braid_PtFcnSpatialNorm spatialnorm, braid_PtFcnBufSize bufsize, braid-_PtFcnBufPack bufpack, braid_PtFcnBufUnpack bufunpack)
- braid_Int braid_TestCoarsenRefine (braid_App app, MPI_Comm comm_x, FILE ∗fp, braid_Real t, braid_Real fdt, braid_Real cdt, braid_PtFcnInit init, braid_PtFcnAccess access, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum, braid_PtFcnSpatialNorm spatialnorm, braid_PtFcnCoarsen coarsen, braid_PtFcnRefine refine)
- braid_Int braid_TestAll (braid_App app, MPI_Comm comm_x, FILE ∗fp, braid_Real t, braid_Real fdt, braid_-Real cdt, braid_PtFcnInit init, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum, braid_Pt-FcnSpatialNorm spatialnorm, braid_PtFcnBufSize bufsize, braid_PtFcnBufPack bufpack, braid_PtFcnBufUnpack bufunpack, braid_PtFcnCoarsen coarsen, braid_PtFcnRefine refine)

10.3.1 Detailed Description

Define headers for XBraid test routines. This file contains routines used to test a user's XBraid wrapper routines one-by-one.

Index